NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93943-5002

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS EC-92-002	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (*If applicable*) 33	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000	7b. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Naval Sea Systems Command 8b. OFFICE SYMBOL (*If applicable*) PMS-412	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*)	10. SOURCE OF FUNDING NUMBERS Program Element No. / Project No. / Task No. / Work Unit Accession Number

11. TITLE (*Include Security Classification*)
(U) A TECHNIQUE FOR PREDICATABLE REAL-TIME EXECUTION IN THE AN/UYS-2 PARALLEL SIGNAL PROCESSING ARCHITECTURE

12. PERSONAL AUTHOR(S) Little, Brian S.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*year, month, day*)	15. PAGE COUNT
Master's Thesis	From To	911201	141

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18. SUBJECT TERMS (*continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUBGROUP	AN/UYS-2, Data-flow Processing, Processing Graph Methodology, Signal Processing, Scheduling, Large-grain Data-flow Architectures

19. ABSTRACT (*continue on reverse if necessary and identify by block number*)

The AN/UYS-2 provides the Navy with a state of the art Digital Signal Processor. The AN/UYS-2 is programmed utilizing the Processing Graph Methodology (PGM), which represents specific tasks as nodes in a graph. It utilizes a simple First-Come-First-Served (FCFS) run-time resource allocation mechanism that supports large-grain data-flow processing. While the mechanism is robust, easy to implement, and results in low run-time overhead, it is difficult to predict if a given PGM will meet the application requirements. Therefore, an approach that uses compile-time analysis to exploit the periodic arrival of data and a priori knowledge of the amount of computation and communication overhead is investigated. Improvement in performance of the machine when the PGM graphs are restructured using this approach, called Revolving Cylinder scheduling, is observed, and it is found to be an effective approach when there is a high communication overhead or when the PGM nodes are of uniform size.

T257806

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS REPORT ☐ DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Shridhar Shukla	22b. TELEPHONE (*Include Area code*) 408-646-2764	22c. OFFICE SYMBOL EC/SH

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Approved for public release; distribution is unlimited.

A Technique for Predictable Real-Time Execution
in the AN/UYS-2 Parallel Signal Processing Architecture

by

Brian S. Little
Lieutenant, United States Navy
B.S.E., University of Florida

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

ABSTRACT

The AN/UYS-2 provides the Navy with a state of the art Digital Signal Processor. The AN/UYS-2 is programmed utilizing the Processing Graph Methodology (PGM), which represents specific tasks as nodes in a graph. It utilizes a simple First-Come-First-Served (FCFS) run-time resource allocation mechanism that supports large-grain data flow processing. While the mechanism is robust, easy to implement, and results in low run-time overhead, it is difficult to predict if a given PGM will meet the application requirements. Therefore, an approach that uses compile-time analysis to exploit the periodic arrival of data and a priori knowledge of the amount of computation and communication overhead is investigated. Improvement in performance of the machine when the PGM graphs are restructured using this approach, called Revolving Cylinder scheduling, is observed; and it is found to be effective when there is a high communication overhead or when the PGM nodes are of uniform size.

Thesis
L6915
c.1

TABLE OF CONTENTS

I. INTRODUCTION 1

A. BACKGROUND 2

1. AN/UYS-2 Design Theory 2

2. AN/UYS-2 Design Problems 4

B. OBJECTIVES 5

C. THESIS ORGANIZATION 5

II. ARCHITECTURE AND PROGRAMMING OF THE AN/UYS-2 7

A. ARCHITECTURE 7

1. Modularity 8

a. The Scheduler 10

b. The Global Memories 10

c. The Arithmetic Processors 11

d. The Input/Output Processors 12

e. The Input Signal Conditioner 12

2. Cohesiveness 12

a. Control Buses 13

MONTEREY, CALIFORNIA 93943-5

b. Data Transfer Network 13

c. Command Program Processor 13

B. Programming 14

1. Graphical Interface 14

2. Graph to Program Conversion 15

III. SCHEDULING OF PGM. ON THE AN/UYS-2 17

A. SIGNAL PROCESSING REQUIREMENTS 17

1. Desirable Characteristics In Execution 18

2. Resource Allocation 18

B. FIRST-COME-FIRST-SERVED SCHEDULING 20

1. Advantages 21

2. Disadvantages 21

3. A Simple Example 22

C. REVOLVING CYLINDER SCHEDULING 23

1. Implementation 25

2. Advantages 28

3. Disadvantages 29

4. A Simple Example 29

IV. THE SIMULATOR 31

A. IMPLEMENTATION 31

1. Communications 34
2. Major Resource Elements 37
a. The Input/Output Processor 37
b. The Global Memories 38
c. The Scheduler 39
d. The Arithmetic Processors 40
B. THE LANGUAGE - C++ 41
C. USER INTERFACE 42
D. LIMITATIONS OF THE SIMULATOR 43

V. PERFORMANCE EVALUATION 44
A. CORRELATOR APPLICATION 44
1. Description 44
2. Output and Interpretation 45
B. CORRELATOR WITH UNIFORM NODE SIZES 49
1. Description 49
2. Output and Interpretation 50
C. CORRELATOR WITH CHAINED NODES 54
1. Description 54
2. Output and Interpretation 55
D. FFT APPLICATION 58
1. Description 58

2. Output and Interpretation 60

VI. CONCLUSIONS 65

A. SUCCESS OF THE RC APPROACH 65

1. Communication Intensive 65

2. Non-Communication Intensive 65

B. IMPROVING RC 66

C. PROPOSED RESEARCH 67

1. Hardware Modifications 67

a. Systolic Array Processor 67

b. Open Architecture 67

2. Software Modifications 67

a. User Friendly Processing Graph Methodology 68

b. Throughput Enhancements 68

(1) Node Chaining 68

(2) Scheduling 68

(3) Fault Tolerance 69

APPENDIX A: REVOLVING CYLINDER CODE 70

APPENDIX B: PGM REPRESENTATION CODE 87

APPENDIX C: MAIN SIMULATOR CODE 90

APPENDIX D: INPUT/OUTPUT PROCESSOR CODE 93

APPENDIX E: GLOBAL MEMORY CODE 96

APPENDIX F: SCHEDULER CODE 99

APPENDIX G: ARITHMETIC PROCESSOR CODE 107

APPENDIX H: INTER-COMMUNICATION CODE 113

APPENDIX I: RESULT GENERATION CODE 115

LIST OF REFERENCES 120

INITIAL DISTRIBUTION LIST 124

LIST OF TABLES

Table I: FUNCTIONAL ELEMENT REQUIREMENTS 9

Table II: EXECUTION OF 36 CYCLES UTILIZING FCFS SCHEDULING . 24

Table III: RC ASSIGNMENT SCHEDULE FOR SAMPLE PGM GRAPH . . . 30

LIST OF FIGURES

Figure 1: The AN/UYS-2 Architecture 9

Figure 2: A Sample PGM Graph 15

Figure 3: Converted PGM Program 16

Figure 4: A Simple PGM Graph 23

Figure 5: An Algorithm to Perform RC Assignment 26

Figure 6: An Algorithm to Assign Dependencies 27

Figure 7: A Possible Restructured PGM 27

Figure 8: Graphical Input Format for the Simulator 32

Figure 9: Data Structure Representation of a PGM Graph 32

Figure 10: The Simulator Structure 33

Figure 11: Graphical Description of Correlator Application 45

Figure 12: Correlator Graph Mean 47

Figure 13: Correlator Graph Mean Blow Up 47

Figure 14: Correlator Graph Standard Deviation 48

Figure 15: Correlator Graph Normalized Output Completion Time 48

Figure 16: Correlator Graph AP Efficiency 49

Figure 17: Correlator Graph Structure with Uniform Sized Nodes 50

Figure 18: Correlator Structure Means for Equal Node Times 51

Figure 19: Correlator Structure Means Blow Up with Equal Times 51

Figure 20: Correlator Structure Deviation with Equal Times 52

Figure 21: Equal Correlator Structure Output Arrival Times 52

Figure 22: AP Efficiency for Correlator Structure Equal Times 53

Figure 23: Correlator Graph Structure with Chained Nodes 54

Figure 24: Correlator Graph Structure Mean with Chained Nodes 55

Figure 25: Correlator Graph Structure Mean Blow Up Chained 56

Figure 26: Correlator Graph Structure Deviation Chained Nodes 56

Figure 27: Correlator Graph Structure Output Times Chained 57

Figure 28: Correlator Graph Structure AP Efficiency Chained 58

Figure 29: FFT Graph Description 59

Figure 30: FFT Mean 61

Figure 31: FFT Mean Blow Up 61

Figure 32: FFT Standard Deviation 62

Figure 33: FFT Standard Deviation Blow Up 62

Figure 34: FFT Normalized Instance Completion Output 63

Figure 35: FFT AP Efficiency 63

LIST OF SYMBOLS

DSP	-	Digital Signal Processing
SEM	-	Standard Electronic Module
FE	-	Functional Element
SCH	-	Scheduler
AP	-	Arithmetic Processor
GM	-	Global Memory
IOP	-	Input/Output Processor
ISC	-	Input Signal Conditioner
CBUS	-	Control Bus
DTN	-	Data Transfer Network
PE	-	Physical Element
PGM	-	Processing Graph Methodology
AU	-	Arithmetic Unit
CU	-	Control Unit
CPP	-	Command Program Processor
FCFS	-	First Come First Served
RC	-	Revolving Cylinder
ASCII	-	American Standard Code for Information Interchange
WQ	-	Write Queue

QOT	-	Queue Over Threshold
QOC	-	Queue Over Capacity
QUC	-	Queue Under Capacity
EIS	-	Execute Instruction Stream
SIS	-	Send Instruction Stream
AIS	-	Accept Instruction Stream
RQ	-	Read Queue
AQ	-	Accept Queue
RFIS	-	Ready For Instruction Stream
CQ	-	Consume Queue
FFT	-	Fast Fourier Transform
2-D	-	Two Dimensional
1-D	-	One Dimensional

ACKNOWLEDGEMENTS

I would like to thank my advisor, Shridhar Shukla, for all of his assistance and contributions throughout the duration of this project. In addition, I would like to express my deepest gratitude to my wife, Joan. Her understanding and patience throughout the long computer nights were invaluable. Without her, none of this would have been possible.

I. INTRODUCTION

Since the advent of radar before World War II, the success of modern warfare has depended on the ability of a system to process "... electronic signals to detect, localize, attack, and counter increasingly sophisticated threats." [RICE 90, pp. 1-2] The more sophisticated the system, the more complex the signal processing requirements. Not only does the complexity depend on the number of operations to be performed, but also on the time interval available for completion [BELLANGER 84, pp. 3]. Within the realm of modern warfare, milliseconds can be the difference between survival or death. The ability to quickly interpret and disseminate incoming target data provides the user a discernible edge during today's modern warfare.

No longer does the Navy need to rely upon electro-mechanical processors. The development of data transducers and the advent of multi-processors allow electrical signals to be processed in real time using digital methods [BELLANGER 84, pp. 4-5]. But, if digital signal processing machines are to operate in real time, they must operate at a rate which is closely related to the sampling frequency of the signals.

The AN/UYS-2 Enhanced Modular Signal Processor is utilized in many different signal processing applications, from the acoustic system on a P3-C "Orion" aircraft to the BSY-2 Sonar Suite on the SSN-21 "Seawolf" class submarine. Although different configurations are possible, the search mission drives the envelope of the processing system. The United States' Navy's signal processing requirements have been increasing

since the advent of electronics conception in the early part of the twentieth century and "... are expected to increase tenfold within the next ten years." [RICE 90, pp. 2]

The AN/UYS-2 is meant to provide the United States' Navy with a standard, programmable, modular, multi-processor capable of meeting the digital signal processing requirements into the twenty-first century. Yet, modifications will be required if it is to maintain its goal.

A. BACKGROUND

The innovation of new weapon platforms, led by Autonomous Underwater Vehicles and Remotely Piloted Vehicles, and the advancement of weapon technology requires an intelligent stand-alone programmable multi-processor.

"To achieve high performance in a processor specialized for signal processing, the need to depart from the simplicity of von Neumann computer architectures is axiomatic." [LEE 87, pp. 24]

1. AN/UYS-2 Design Theory

Parallel computations that exist in signal processing can be naturally represented as data-flow graphs. The data-flow approach was first presented by Karp, and it has since been expanded by many including Dennis and Watson [KARP 66, pp. 1390-1411, DENNIS 80, pp. 48-56, WATSON 82, pp. 51-57]. These graphs not only describe the dependencies between different parts of the computation required in an application, but also provide built-in scheduling and synchronization.

While data-flow techniques have been applied to digital signal processing since its earliest days, Navy sensor systems have continued to employ a control-flow method until the AN/UYS series development during the 1980's [LEE 90, pp. 333, RICE 90, pp. 2]. Time-line control, in which a single control signal generates program execution and provides the output, characterizes a control-flow architecture. Multi-thread control flow architectures are achievable by use of more than one control signal, though it is difficult to develop programs that mold themselves to this simplified structure.

Data-flow representation of digital signal processing algorithms provides a natural exploitation of concurrence [LEE 90, pp. 333]. Typical data-flow algorithms execute a task based upon the availability of input data and machine resources, thereby enabling the data to exist only between its production and consumption and eliminating the need for a distinct program counter [RICE 90, pp. 2].

A distinction must be made between *large-grain* and *fine-grain* data-flow architectures. *Fine-grain* architectures have their uses in Very Large Scale Integration as documented by Koren, Silberman, and Dennis [DENNIS 80, pp. 48-56, KOREN 83, pp. 335-337]. But, the use of *fine-grain* data-flow within the modular design of the AN/UYS-2 would produce unreasonable communication overheads. Therefore, the AN/UYS-2 is built around the *large-grain* data-flow architecture approach. Due to the generality of the data-flow paradigm, it can be used to specify and exploit the parallelism at the instruction level as well as at the task level [BROBST 87, pp. 40-45, SAWKAR 83, pp. 344]. The theoretical foundation for consistency of such representations has been well studied by Lee, Karp, and Miller [LEE 87, pp. 24-35, KARP 66, pp. 1390-1411].

The focus of this work is on task-level parallelism in such applications expressed using data-flow graphs. Such computations are also classified as *pipelined function-parallel computations* and *synchronous data-flow computations* [LEE 87, pp. 24-35]. However, the machine must provide mechanisms to manage the data that flows through the graph and to capture the intrinsic scheduling and synchronization.

The AN/UYS-2 utilizes a distributed run-time operating system which implements a hybrid control-flow/data-flow architecture by utilizing the data-flow technique at the task level and the control-flow approach at the elementary processing level [POPS 89, pp. 2-9, RICE 90, pp. 2]. This helps to minimize the communication costs involved while providing efficient elementary level execution.

2. AN/UYS-2 Design Problems

The mechanisms involved in managing graph data flow, intrinsic scheduling, and synchronization, typically operating at run-time, result in overheads that lead to suboptimal performance.

The non-deterministic (first-come-first-served) scheduling strategy used by the AN/UYS-2 will not be able to maintain throughput with the rapidly increasing data-flow graph bandwidth.

Increasing sensor system complexity compounded by decreasing allowable reaction time is likely to degrade the AN/UYS-2 capabilities considerably.

B. OBJECTIVES

Since the AN/UYS-2 utilizes a non-real-time strategy to schedule the application's nodes to free processors, this thesis investigates a robust, compile-time technique that supports a simple run-time mechanism to improve throughput and predictability in the AN/UYS-2 architecture including:

- Investigation of Digital Signal Processing (DSP) requirements
- Development of a compile-time algorithm for graph restructuring
- Design and implementation of an AN/UYS-2 software simulator
- Performance evaluation and comparison of the existing and proposed scheduling algorithms [POPS 90, pp. 6-3].

A compile-time approach is possible in DSP due to the tremendous amount of information that is known about each task. It simplifies the application developer's task and results in no change to the run-time mechanism.

C. THESIS ORGANIZATION

Chapter II describes the architecture of the AN/UYS-2 in detail and discusses the program interface. The modular design and actual components are elaborated. Chapter III studies the requirements of signal processing applications, examines the current AN/UYS-2 scheduling strategy, and proposes a deterministic scheduling approach. C++ code for the deterministic algorithm is provided. A simple graph is scheduled as an example. Chapter IV describes the simulator constructed for this thesis and its limitations. Background information and simulator specifics, including coding techniques, are also provided. Chapter V analyzes the performance of the non-deterministic and

deterministic algorithms by utilizing two key signal processing examples. Chapter VI provides a summary of the overall work, and gives recommendations for future expansion and improvement.

II. ARCHITECTURE AND PROGRAMMING OF THE AN/UYS-2

This chapter describes the architecture of the AN/UYS-2 in detail and discusses how the AN/UYS-2 is programmed to achieve its desired results. The data-flow principle is examined from the view point of the modular components.

A. ARCHITECTURE

Different DSP applications have specific processing requirements. The Navy's diverse operating environment, stringent operational tempo, and sundry weapon platforms contribute heavily towards the requirement for a modular based design. In addition, specific reliability criteria required in today's weapon systems are easier to implement and maintain in a modular system. Within the last twenty years, "... substantial reductions in the cost of digital computation have occurred accompanied by an improvement in performance, speed, memory capacity, and ease of programming ..." making modular design even more attractive [BEAUCHAMP 79, pp. iv]. Rapidly changing hardware technology and DSP requirements are easier to implement in a modular design. The AN/UYS-2 architecture provides for many modular machine configurations, tailoring itself to a set application, while maintaining a cohesiveness and implementation ease as an overall system. [POPS 90, pp. 2-6].

The AN/UYS-2 comes packaged in two Standard Electronic Module (SEM) type implementations which enable the unit to be configured for either ship based or aircraft mounting. The standard SEM "B" cabinet format and the smaller SEM "E" format which

optimizes the unit with regards to size and weight, making it ideal for aircraft dissemination. Within each SEM class, the AN/UYS-2's components are constructed on very high speed integrated circuit cards. Specific hardware implementation is documented by Rice. [RICE 90, pp. 4-7]

1. Modularity

The AN/UYS-2's modular design is based on six functional element (FE) types: the Scheduler (SCH), the Arithmetic Processors (AP's), the Global Memories (GM's), the Input/Output Processors (IOP's), the Command Program Processor (CPP), and the Input Signal Conditioner's (ISC's); and two data paths: the Control Bus (CBUS) and the Data Transfer Network (DTN) as shown in Figure 1. These six FE types can be fused into any combination as long as the requirements specified in Table I are maintained, thereby enabling the AN/UYS-2 to be tailored to a specific application [APPLIC 90, pp. 2-3]. Each FE performs certain system tasks. The AN/UYS-2 layout enables the incorporation of new FE's into the modular structure as long as the communication interface to the DTN and CBUS remains constant.

These modular FE's perform parallel computation by associating "... each step of the algorithm with a node of a directed graph." [KARP 66, pp. 1390] As data for these nodes becomes available, each node must be scheduled to execute on an AP. This is accomplished by the scheduler. The actual programming approach is discussed later in this chapter.

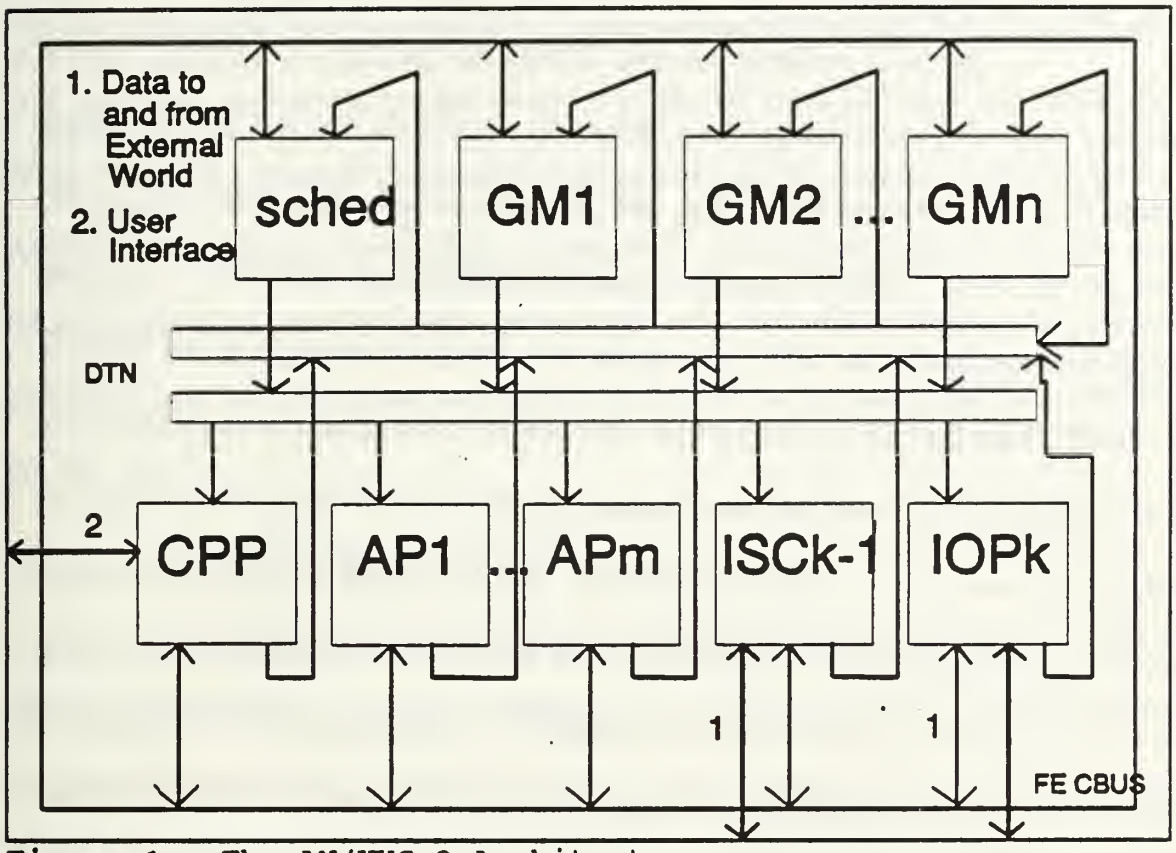

Figure 1: The AN/UYS-2 Architecture

Table I: FUNCTIONAL ELEMENT REQUIREMENTS

	Minimum	Maximum
Number of AP's	1	59 - IOP's - ISC's
Number of GM's	1	30
Number of SCH's	1	1
Number of IOP's	1 OR	59 - AP's - ISC's
Number of ISC's	1	59 - AP's - ISC's
Number of CPP's	1	1
Total Number	5	63

a. The Scheduler

The SCH performs the node scheduling operation of matching a ready node to a free AP by maintaining four tables: the ready-node list, the free processing element (PE) list, the node status table, and the queue-to-node table [RICE 90, pp. 4, POPS 90, pp. 6-3]. The SCH receives queue information from the GM's. As queues exceed threshold levels, the GM's send queue over threshold messages to the SCH via the CBUS. Since the SCH also receives AP availability information via the CBUS, when all of a graph node's queues are over threshold, it then attempts to match free AP's to ready graph nodes. If a match is successful, and that graph node is not currently executing, scheduling data is sent to the GM's in the form of a message via the Control Bus (CBUS); and the database tables are updated to reflect the match. If a match is unsuccessful, the node status table and ready node list are updated to indicate the new ready node. When an AP indicates to the SCH that it is Ready For Instruction Stream (RFIS), the SCH checks the ready node list for nodes ready to execute and assigns one to the now available AP. [POPS 90, pp. 6-3]

b. The Global Memories

The GM provides the data storage for the AN/UYS-2 and executes memory management primitive functions. Each Processing Graph Methodology (PGM) queue, discussed later in this Chapter, is allocated to a GM for storage, and that GM maintains that queue's state information. Queue state information consists of read, produce, consume, and threshold quantities, queue size, and a queue identification number. When a queue exceeds threshold or capacity values, or returns under threshold,

the GM's notify the SCH via the CBUS. Upon receiving instructions from the SCH, the GM's send the appropriate *execute* messages to the AP's and IOP's via the CBUS and DTN. The GM's maintain the control variables and node information necessary for the AP's to execute the node successfully. After the AP has completed node breakdown and informed the GM, the GM consumes the input data involved by freeing the storage previously used by those queues and notifies the SCH of the current queue level. [POPS 90, pp. 7-9]

c. The Arithmetic Processors

After the SCH sends the node *execute* instruction to the GM's, the GM's relay the information, by another message via the CBUS, to an AP. The AP then executes the actual signal processing primitives. The message from the GM's contains all of the required information for the AP to read the necessary queue data from the GM's (set-up), execute the designated primitives, and write the output queue data back to the GM's (breakdown). The AP's consist of an Arithmetic Unit (AU) and a Control Unit (CU); consequently, three nodes can be associated with an AP at any one time:

- The first node being setup within the CU
- The next node executing within the AU
- The third node performing breakdown within the CU.

When the AP has completed execution of its current node and setup of its next node, it notifies the SCH via the CBUS that it is ready to process the following node. The ability of the CU to concurrently perform setup and breakdown on distinct nodes and to notify the SCH without the AU's knowledge helps to increase the concurrence involved with

minimum loss of through-put due to the non-availability of resources. [RICE 90, pp. 5, POPS 90, pp. 9-8]

d. The Input/Output Processors

The IOP's provide for raw digitized sensor data input and processed data output from the AN/UYS-2 by buffering and formatting the input and output data to synchronize the different external device data rates with the internal network. Upon arrival of sufficient external data as determined by input buffer size, the IOP's dispatch the input data to the GM queues after an amount specified by the application programmer is received. The IOP's receive the output data from the GM's and redirect it to the external world. [POPS 90, pp. 8-1]

e. The Input Signal Conditioner

In addition to performing the same functions as the IOP's, the ISC's perform signal conditioning operations to reduce input data bandwidth and generates output for sensor control. The ISC's are capable of performing analog-to-digital conversions during the input/output process. [POPS 90, pp. 10-1]

2. Cohesiveness

Despite the modularity of the AN/UYS-2, key elements of its structure mold it into a cohesive unit. The cohesiveness is built around the ability of the modules to communicate and interact with each other. The communication among FE's occurs over the Data Transfer Network and Control Buses. All interaction between the user and the AN/UYS-2 occurs in the Command Program Processor.

a. Control Buses

The FE built-in-test control bus (BITCBUS) provides a means with which the Command Program Processor (CPP) can test the AN/UYS-2 system, and the CBUS provides the main means by which the modular FE's communicate short messages of data-flow control information [RICE 90, pp. 5].

b. Data Transfer Network

The DTN provides for up to 16 simultaneous, asynchronous, unidirectional 32-bit data transfers among FE's by continuously polling the data sources. When a source requests a transfer to a destination that is not already receiving data, the DTN path is established and the transfer conducted. Possible DTN configurations include 2, 4, 8, or 16 input and output ports; each port can have up to four further disseminations. [POPS 90, pp. 4-1]

c. Command Program Processor

The CPP provides the glue that holds the entire AN/UYS-2 architecture together. The CPP acts as the overall control unit for the AN/UYS-2 by performing the following functions:

- Data-flow graph management
- Tactical interface
- Input/output configuration control
- System performance monitoring [POPS 90, pp. 5-1].

B. Programming

New programming methodologies are required for data-flow architectures because data-flow computers depart from conventional architectures [DENNIS 83, pp. 331]. The AN/UYS-2 is programmed using the Enhanced Modular Signal Processor Common Operational Support Software, which refers to the implementation of the PGM on the AN/UYS-2 [POPS 90, pp. 2-8, ECOS 89, pp. 1-35].

1. Graphical Interface

The PGM yields a convenient means of writing application software without concern for the specific architecture of the machine on which it would run [PGMTUT 90, pp. 1-1]. The PGM provides the application programmer with a high level graph oriented language, provides a method of translating these graphs into load modules that the AN/UYS-2 can recognize, and furnishes a run-time support environment which expands graph realizations and manages execution of graph instances [APPLIC 90, pp. 2-5].

A PGM application consists of a *directed, acyclic* graph with nodes representing large grain computations called *primitives*, which are chosen from a self-contained library of signal processing functions. A simple example PGM appears in Figure 2. The edges of the graph represent queues which receive data from the source primitive and supply data to the destination primitive. Conversion of the PGM description into a executable data-flow graph entails only specifying the read, produce, consume, and threshold values. The AN/UYS-2 is designed to run several signal processing applications simultaneously. Therefore, several instances of multiple PGM's

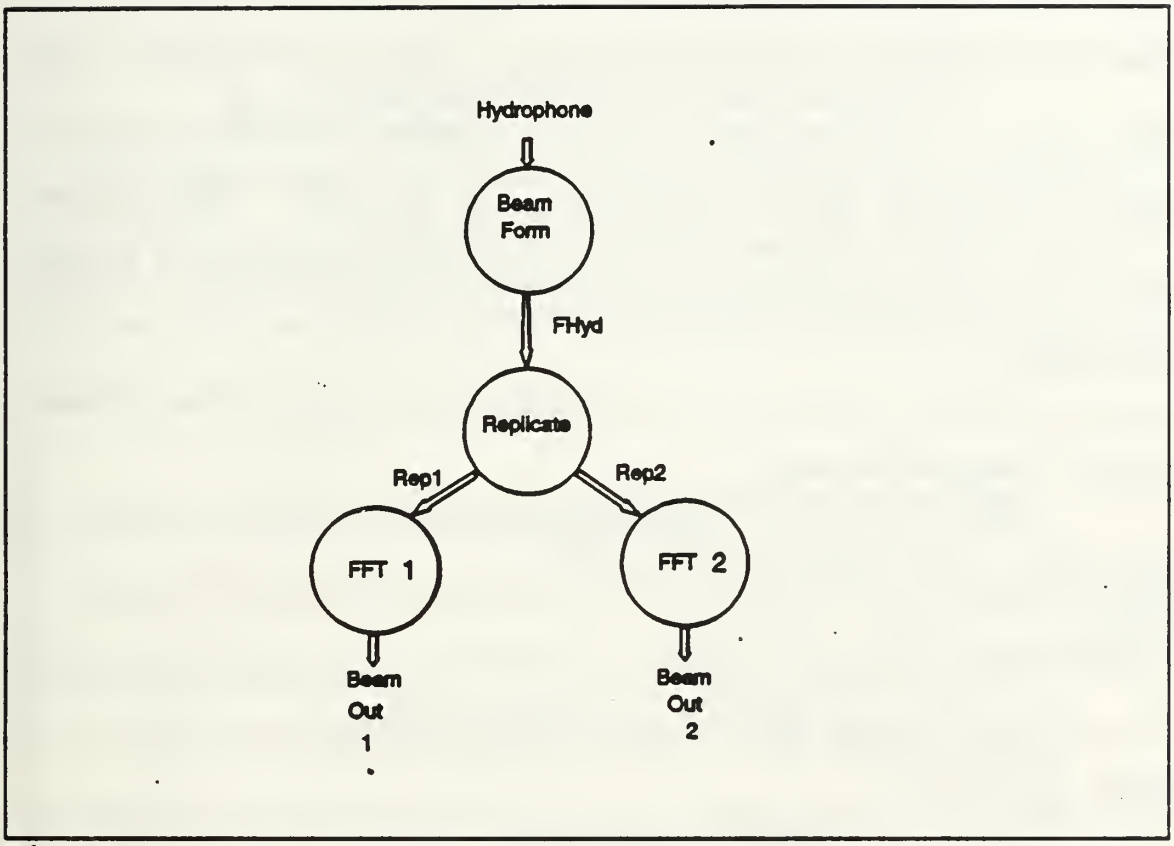

Figure 2: A Sample PGM Graph

may be running at the same time. The graph is loaded into the AN/UYS-2 by the CPP prior to graph initialization.

2. Graph to Program Conversion

Since the AN/UYS-2 is unable to recognize the PGM graph directly as shown previously in Figure 2, the graph must be converted to a *key-word* program. This conversion is typically performed by machine utilizing the guidelines supplied in the Application Programmer User Manual [APPLIC 90, pp. all]. A possible program for the graph of Figure 2 appears in Figure 3. The node primitives that appear within the nodes are documented in the primitive library [PRIMLIB 90, pp. all]. The name of the graph

is specified along with its external input and output queues. Constants are then initialized followed by a description of all queues and nodes. A queue is described by a name followed by a ":" and a type name. A node is completely described by a name, primitive, input queues "prim_in", and output queues "prim_out." The constructed program is then loaded into the AN/UYS-2 and executed by the CPP following the data-flow paradigm.

```
%graph (SamplePGMGraph
        INPUTQ=Hydrophone : fixed(1)
        OUTPUTQ=BeamOut1,BeamOut2 : int)
%GIP scan : int initialize to 4096
%Queue (FHyd : dfloat)
%Queue ([1..2]Rep : dfloat)
%Node (BeamForm
       primitive=BFR_FREQ
       prim_in=scan,Hydrophone threshold=scan
       prim_out=FHyd)
%Node (Replicate
       primitive=DFC_REP
       prim_in=1,2,FHyd threshold=1
       prim_out=[1..2]Rep)
%Node (FFT1
       primitive=FFT_CR
       prim_in=scan,[1]Rep threshold=scan
       prim_out=BeamOut1)
%Node (FFT2
       primitive=FFT_CR
       prim_in=scan,[2]Rep threshold=scan
       prim_out=BeamOut2)
%endgraph
```

Figure 3: Converted PGM Program

III. SCHEDULING OF PGM ON THE AN/UYS-2

This chapter diagrams the specific scheduling of PGM on the AN/UYS-2 by examining the signal processing requirements and the methods of resource allocation. The current First-Come-First-Served (FCFS) implementation and the proposed *real-time* scheduling algorithm are examined in detail.

A. SIGNAL PROCESSING REQUIREMENTS

Due to DSP's concurrency and high throughput requirements, large-grain data-flow programming models can be used effectively to exploit the intrinsic parallelism [PARHI 88, pp. 178]. General data-flow processing requires a direct hardware implementation of the data-flow paradigm [GURD 85, pp. 34-52, BROBST 87, pp. 40-45]. This results in unmanageable overheads. However, for specific classes of applications, such as signal processing, data-flow can be managed very efficiently at the macro level to obtain significant performance improvement. This is due to the ability of representing digital signal processing applications as synchronous data-flow graphs [LEE 87, pp. 24].

Executing data-flow descriptions of DSP's applications on parallel processors requires decisions about allocation, ordering, and data movement. In the AN/UYS-2 context, allocation refers to the assignment of PGM nodes to resources. GM's and AP's constitute the majority of the resources. Ordering relates the node assignment and execution sequence on these resources. Data movement reflects the method and amount of data internally transferred in between executions. Since large-grain data-flow

architectures inherently possess a high decision making overhead, the data-flow principle must be allowed to take effect in order to capture the concurrency and minimize the decision making overhead. Synchronous applications assist in limiting this overhead. The three properties of these applications that make this possible are availability of *a priori*, knowledge of the amount of data produced and consumed, known function execution times, and negligible use of conditionals and recursion. [LEE 87, pp. 25-31]

1. Desirable Characteristics In Execution

In real-time signal processing applications, the principle requirements are predictability and performance as measured by throughput. An additional characteristic imposed by the AN/UYS-2 is on-line reconfiguration. Since the AN/UYS-2 is a data-flow architecture, this predictability is critical in Navy sensor systems. Given the non-determinism of large-grain data-flow model and a set number of available resources for computational assignment, how can deterministic, rate-optimal through-put be guaranteed? While this question has spawned complete design environments like *Gabriel*, designed by Lee, the answer undoubtedly lies in resource allocation [LEE 89, pp. 333-335]. Efficient resource allocation and low communications overhead lead to the high through-put, deterministic output required by DSP.

2. Resource Allocation

Resource allocation forms the basis of the system designer's dilemma. Based on how a graph node and arc attributes are used at compile time and how much control information is generated to aid the run-time mechanisms, data-flow scheduling

implementations can be classified over a spectrum that ranges from *fully-dynamic* through *fully-static*. Although allocation can be *fully-dynamic*, *self-timed*, *static*, or *fully-static*, typical designs utilize a combination of these allocation methods. *Fully-dynamic* allocation performs all scheduling of nodes at run time based on the readiness of inputs. In *self-timed* allocation systems, the compiler determines the order of node execution and allocates resources, but the firing is determined at run-time by data arrival. *Static* node allocation involves the assignment of a node to a processor, but the order of execution is left up to the run time scheduler based on the node's inputs. In *fully-static* allocation, the compiler determines the exact firing time, assignment, and ordering of nodes based on that node's predicted behavior. [LEE 90, pp. 333-334]

Quasistatic scheduling methods are based on ordering memory accesses by blending the static and self-timed methods into a hybrid solution that supports Von Neumann Architectures without the need for specifically designed data-flow machines. *Quasistatic* scheduling maintains the ordering of nodes on processors while preserving the ordering of accesses to other shared system resources. Since *quasistatic* scheduling incorporates self-timed methods, node execution time can vary without affecting the results. Lee et al. have proposed utilizing hardware semaphore to overcome the high communication overheads generated by *quasistatic* methods [LEE 90, pp. 334-338].

Many different processor allocation schemes have been proposed that are inadequate for actual data-flow systems [DAVIS 79, pp. 1079-1086, ARVIND 80, pp. 7-14, MUNDELL 81, pp. 156-157]. Systems like *Gabriel*, characterized by Lee, utilize compile time static resource assignment and work well on sequential data-flow graphs,

whose very nature enables nodes to be scheduled at compile time rather than run time, relying on the graph structure for enforcement. Periodic Admissible Parallel Schedule (PAPS), also proposed by Lee, assumes sequential data-flow and establishes precedence links between nodes in the graph and synchronizes these links at every graph instance [LEE 87, pp. 25-28]. *Quasistatic* methods ensure run time enforcement by utilizing dynamic control [LEE 87, pp. 25-28]. While algorithms proposed by Ho and Irani maintain concurrency, they do not guarantee deterministic throughput [HO 83].

As mentioned previously, the AN/UYS-2 possesses several sets of resources: the AP's, GM's, IOP's, ISC's, DTN, and CBUS. IOP's and ISC's are hardwired into a certain configuration for the external world. GM's are allocated at compile time when the graph is initiated, and are assigned based solely on the graph queue structure with all input queues associated with a node being assigned to the same GM if possible. AP's are allocated to execute specific graph nodes by the scheduler as they become ready on a first-come-first-served basis at run time. The DTN and CBUS are configured and hardwired for its specific structure during assembly, but individual messages are not allocated to it until run time. [POPS 90, pp. 3-8]

B. FIRST-COME-FIRST-SERVED SCHEDULING

The simple nature of the SCH is motivated by the requirements that it should not be a bottleneck, that it should maintain a high and balanced AP utilization, that it should incorporate multiple applications simply, and that it should behave well during reconfiguration. The SCH implements First-Come-First-Served (FCFS) scheduling by

maintaining a database consisting of PGM graph node and AP information. Available AP's are matched with ready nodes on a continuing basis, interrupted only by user reconfiguration of the system. If the SCH is unable to execute a match, either due to no free AP's or no ready nodes, then the SCH maintains a free AP list and a ready node list. The SCH matches the elements from the lists together in the same FCFS manner as they become available.

1. Advantages

The simplicity of FCFS scheduling earns it the designation as the most attractive scheduling algorithm. The FCFS scheduling algorithm's low *run-time* overhead costs also lends itself to the AN/UYS-2. This simple algorithm ensures close to the maximum possible AP utilization since nodes are sent to AP's as soon as they become available.

2. Disadvantages

The major disadvantage within the AN/UYS-2 scheduling arises from unpredictability in EMSP output arrival. The dynamic assignment of AP's to ready nodes suffers from the following intrinsic disadvantage when data arrives periodically from the external world. The nodes that depend only upon the receipt of external data get ready for execution, and therefore enter the ready list at a rate which is independent of the other nodes. If the graph latency is longer than the data arrival period, nodes in the lower portion of the graph get ready after nodes in the upper portion. Since the machine follows a FCFS node to AP assignment strategy, the top nodes execute at a higher rate

than the nodes lower down in the graph. The lower graph nodes' execution will catch up only as the upper nodes' output queues exceed capacity and prevent the top nodes from entering the ready list. As a result, processed output data arrives unpredictably leading to the possibility of intolerable delays and insufficient buffer space especially under high loads. This non-uniformity in output arrival arises because the task-level data-flow mechanism does not allow any mechanism to control the input nodes [SHUKLA 91, pp. 222-231]. It will be present when the resource allocation at all stages of the parallel machine is not coupled to the input data arrival and has been observed due to FCFS scheduling [SHUKLA 91, pp. 222-231].

3. A Simple Example

The PGM graph shown in Figure 4, a simple graph with six nodes whose AP execution time are shown inside the vertices, can be used to demonstrate FCFS scheduling. For simplicity, we neglect the set-up and break-down times associated with each node. Consider that an AN/UYS-2 with two AP's, each of which has a processing speed of one unit, should be able to attain a maximum data rate and start a new graph instance every:

$$\frac{TotalExecutionTime}{NumberofAP's} = \frac{12}{2} = 6Cycles. \quad (1)$$

Assume, for simplicity of explanation, that data arrives at this exact rate. A possible resulting order of execution utilizing FCFS scheduling is documented in Table II. The non-uniformity that exists in the instance completion times is inherently obvious in this table by examining the clock cycles for node "f." It should be noted that this is only one

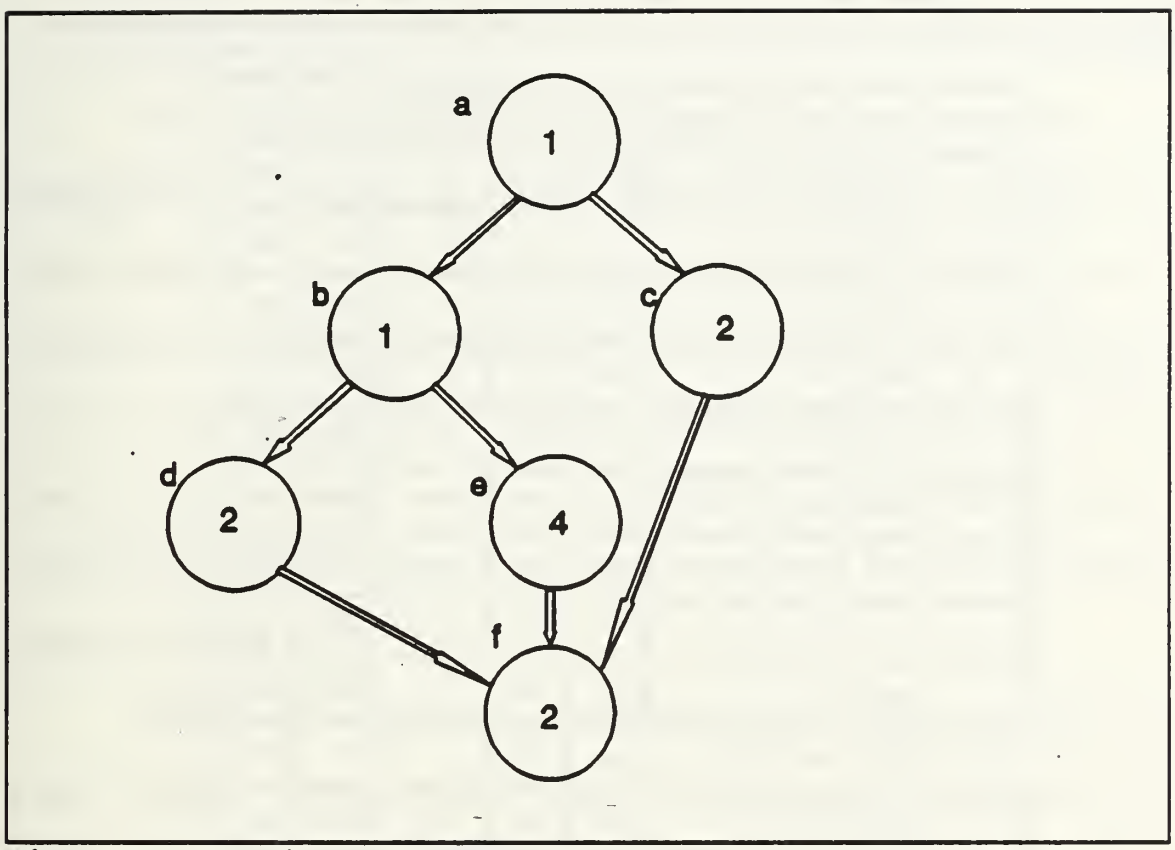

Figure 4: A Simple PGM Graph

possible FCFS scheduling order. Different orders are possible due to set-up and breakdown delays affecting the node arrival order at the SCH.

C. REVOLVING CYLINDER SCHEDULING

While the simple dispatcher works quite well as a run-time mechanism, it does not yield easily to compile-time analysis. This technique restructures the graphical application description by performing compile-time analysis of the application execution profile. Given a graph, it is possible to systematically determine whether it can be mapped on a certain number of AP's while satisfying the required data rates. It can be proved that the graph can be scheduled (ignoring overheads) such that the consecutive graph instances are

Table II: EXECUTION OF 36 CYCLES UTILIZING FCFS SCHEDULING

Cycle	AP1	AP2	Cycle	AP1	AP2	Cycle	AP1	AP2
1	a1	L	13	b3	e2	25	e4	b5
2	b1	c1	14	d3	c3	26	c5	e5
3	d1	c1	15	d3	c3	27	c5	e5
4	d1	e1	16	e3	f2	28	d5	e5
5	L	e1	17	e3	f2	29	d5	e5
6	a2	e1	18	e3	a4	30	f5	f4
7	b2	e1	19	e3	b4	31	f5	f4
8	c2	f1	20	f3	d4	32	a6	L
9	c2	f1	21	f3	d4	33	c6	b6
10	d2	e2	22	e4	c4	34	c6	e6
11	d2	e2	23	e4	c4	35	d6	e6
12	a3	e2	24	e4	a5	36	d6	e6

separated by a time equal to the *total execution time* of the PGM divided by the number of AP's. Since the AP's will be fully utilized, this time corresponds to the maximum throughput rate. The key idea in the Revolving Cylinder (RC) assignment is that inserting delays in the PGM can produce a graph with better throughput. The idea behind RC scheduling can be traced back to algorithms for overlapping complex operations on pipelined processors [RAU 82, pp. 131-139, SHUKLA 91, pp. 222-231].

1. Implementation

RC scheduling recommends when a graph node should be scheduled, but choosing the AP to schedule it on is left to the run-time dispatcher. This enables the actual scheduling to remain *dynamic*. Assume that there is a cylinder whose circumference is the sum of all of the nodes' execution times divided by the number of AP's in the AN/UYS-2 structure. The idea is to schedule the graph such that it wraps around the cylinder, thereby causing its end to meet its beginning. The separation of beginning from end has the effect of a *divide-by-circumference* counter every time the beginning meets the end.

Each slot in the cylinder is of width equal to the smallest size node in the graph. For each node in the graph, starting with the top and working towards the bottom, attempt to schedule the node at its earliest start time. If it can not be inserted at that time, delay the start time by the width of a slot and repeat until it can be inserted. Adjust the earliest start time of all descendants of that node and repeat the above sequence with the next node as the top node in the graph. This ensures that maximum cylinder usage will result when the cylinder is filled by these algorithms shown in Figure 5.

Once all nodes have been inserted into the cylinder and the cylinder is full, assign dependencies to the nodes based upon their location in the cylinder. For each entry relegated to an AP in the cylinder, if there are other nodes assigned to the same AP with the same index and the node higher up in the cylinder is not an ancestor of the other, then create a dependency from the higher node to the lower. This algorithm is shown in Figure 6. Figure 7 shows a possible restructuring resulting from these

```
procedure Assign_RC(G,P);  /*G is a directed acyclic graph*/
                           /*P is the number of AP's*/
   q ← topological sort(G);/*O(e), q is a queue*/
   for all nodes n_i
      est(n_i) ← 0;        /*est is the earliest start time*/
   circumference ← 0;
   for all nodes n_i
      circumference ← circumference + w(n_i);
      /*w(n_i) is the size of the node n_i*/
   circumference ← ⌈(circumference/P)⌉;
   while q is not empty
      temp ← remove_top(q);
      if sufficient space available in cylinder
         t ← schedule_node(temp, est(temp), cylinder);
         for all descendants of temp
            est(descendent) ←max(est(descendent),t+w(temp);
      else
         cylinder ← increase_cylsize(cylinder,circumference,w)
      end(if)
   end(while)
   headofdepqs ← create_deps(cylinder,circumference,w);

procedure schedule_node(temp,t,cylinder)
   scheduled ← false;
   while not scheduled
      Attempt to insert at time t' = t mod circumference;
      /*insert if space available at that time in cylinder*/
      if inserted
         schedule ← true;
      else
         t' ← (t' + w(temp)) mod circumference;
   end(while)
   return t';
```

Figure 5: An Algorithm to Perform RC Assignment

algorithms in the graph of Figure 4. It should be noted that different schedules which sustain the maximal load could be obtained for any graph. Not all of the dependency delays represented by the slim arrows will be generated by the algorithms. Any assignment of nodes on the surface of the cylinder such that no node is preempted, and no two nodes are mapped to the same square is valid. Actual code implementation for the RC algorithms presented is given in Appendix A.

```
procedure Create_deps(cyl,circum,width);
   for all AP(i)'s
      t ← 0;
      while t < circum
         t' ← 0;
         while t' < circum
            if ((index(cyl[i][t]) = index(cyl[i][t'])) and
               (cyl[i][t] is not an ancestor of cyl[i][t'])
               and (a dependency does not already exist here))
               add a dependency for cyl[i][t] to cyl[i][t'];
            t' ← t' + width;
         end(while)
         t ← t + width;
      end(while)
   end(for)
   add a dependency from a cyl[i][circum] to every input node
   return pointer to the head of the list of dependencies
```

Figure 6: An Algorithm to Assign Dependencies

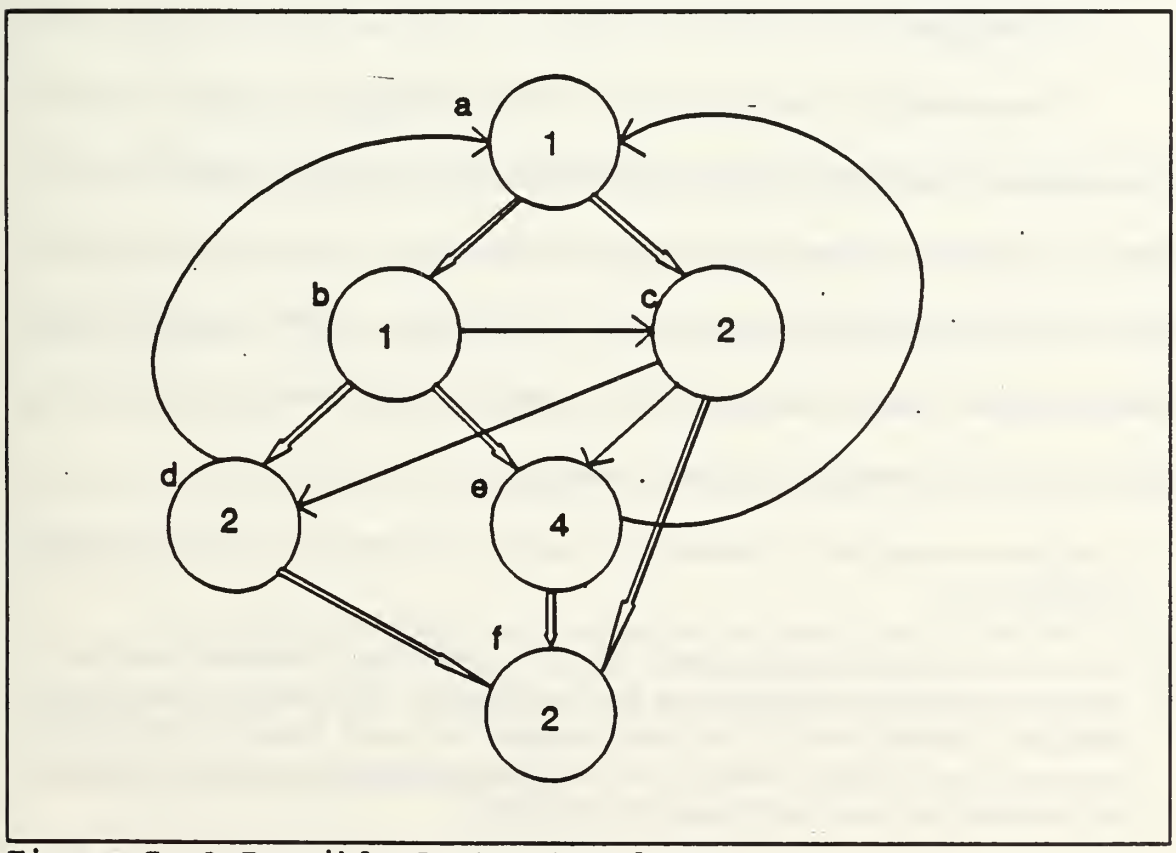

Figure 7: A Possible Restructured PGM

Implementation of the RC database and its incorporation into the AN/UYS-2 requires minimal code modification. The only addition to the SCH code involves function calls to set, adjust, and verify the *tokens* used to represent the dependencies. When the originating node of a dependency is due to be scheduled, the *token* associated with that node in the *dependency list* is incremented one unit. Conversely, when the destination node of a dependency is ready to be scheduled, the *token* with that node in the *dependency list* is decremented one unit. *Forward* graph dependencies are initially set to zero, and *backward* graph dependencies are initially set to one. This enables the initial nodes to execute at least once prior to being inhibited by dependencies. The *token* code is provided in Appendix A and the SCH code modifications in Appendix F.

2. Advantages

The availability of multiple schedules which could sustain the same throughput has an important advantage with respect to the AN/UYS-2: nodes can be grouped together on the surface of the cylinder so as to introduce optimization to minimize the loss of AP cycles due to such overheads as *set-up* and *break-down* times. There are several other advantages of this node-AP assignment if a compile-time technique can be found to enforce it on the scheduler run-time mechanism:

- Compile-time analysis of whether the machine will meet the required data rate becomes easy.
- Since the nodes are associated with AP's at compile-time, it becomes possible to take into consideration optimization such as chaining. For example, although it is possible to assign nodes in the above example in several ways, the assignment shown enables chaining nodes {a,b,c,d} together and chaining nodes {e,f} together to minimize the set-up and breakdown overheads.

- Once it has been determined which nodes are to be chained, the data queues can be allocated to GM's so that the GM contention is minimized.
- There is no non-uniformity in the output generation.

3. Disadvantages

Since RC scheduling keeps track of more node relationship information, there is an immediately higher overhead in this area. But, this overhead can be absorbed by node chaining as discussed in Chapter VI. The requirement for the scheduler to support on-line reconfiguration, typically performed by the operator by replacing one or more branches of the PGM graphs, is difficult to implement without significantly increasing the complexity of the RC algorithms. Yet, this appears to be feasible at the macro-language level and is an important aspect of this approach deeming further investigation.

4. A Simple Example

A RC schedule for the graph of Figure 6 is shown in Table III. After every six clock cycles, another instance of the modified graph can be overlapped with the preceding instance. Since the latencies for the RC algorithm are absorbed by the during the first six cycles of execution as the cylinder fills, the sporadic latencies shown previously in Table II for the FCFS assignment are eliminated and a uniform output rate is generated.

The remainder of this thesis concentrates on developing a simulator which implements the AN/UYS-2 architecture, and on analyzing the performance of the Revolving Cylinder assignment algorithm.

Table III: RC ASSIGNMENT SCHEDULE FOR SAMPLE PGM GRAPH

Cycle # (i ≥ 1)	AP1	AP2
6i - 5	a(i)	e(i-1)
6i - 4	b(i)	e(i-1)
6i - 3	c(i)	f(i-1)
6i - 2	c(i)	f(i-1)
6i - 1	d(i)	e(i-1)
6i	d(i)	e(i-1)

IV. THE SIMULATOR

This chapter examines the structure of the AN/UYS-2 simulator including the language chosen and method of implementation.

A. IMPLEMENTATION

Before the simulator could be developed, the method of PGM representation for the simulator needed to be determined. Implementation of PGM on the simulator consists of maintaining only the key elements of the graph. The actual operation of the AN/UYS-2 in terms of DSP was not implemented. Figure 8 provides the recognizable simulator input, neglecting the columnar headings, for the PGM graph of Chapter III, Figure 4. The graphical description should be located in an American Standard Code for Information Interchange (ASCII) file 'graph.' Data structure for the graph of Figure 8 consists of two parallel queue structures: a *node* queue, which contains pointers to its input and output queues, and a *queue* queue thereby yielding the *tree-like* structure shown in Figure 9. The *gnodes*, 'a' through 'f', shown in Figure 9 contain all of the node information listed in Figure 8, and the *gqueues*, '1' through '9', contain all of the queue information documented in Figure 8. The arrows shown in Figure 9 represent *pointers* which relate the graph infrastructure. Actual code implementation for input of the graphs and some of the necessary simulator access to the graphical data is documented in Appendix B.

9 Constitutes the total number of queues

Queue ID	Node In	Node Out	Arrival Period	Threshold Value	Production Value	OverCapacity Value
1	-1	a	6	1024	1024	8192
2	a	b	0	1024	1024	8192
3	a	c	0	1024	1024	8192
4	b	d	0	1024	1024	8192
5	b	e	0	1024	1024	8192
6	c	f	0	513	513	4096
7	d	f	0	1024	1024	8192
8	e	f	0	1024	1024	1024
9	f	-1	0	1024	1024	1024

6 Constitutes the total number of nodes

Node ID	IOP Node	AIS Size	Execution Time	Number of In Queues	Input Queue ID	Number of Out Queues	Output Queue ID
a	1	256	1	1	1	2	2 3
b	0	256	1	1	2	2	4 5
c	0	256	2	1	3	1	6
d	0	256	2	1	4	1	7
e	0	256	4	1	5	1	8
f	1	256	2	3	7 8 6	1	9

Figure 8: Graphical Input Format for the Simulator

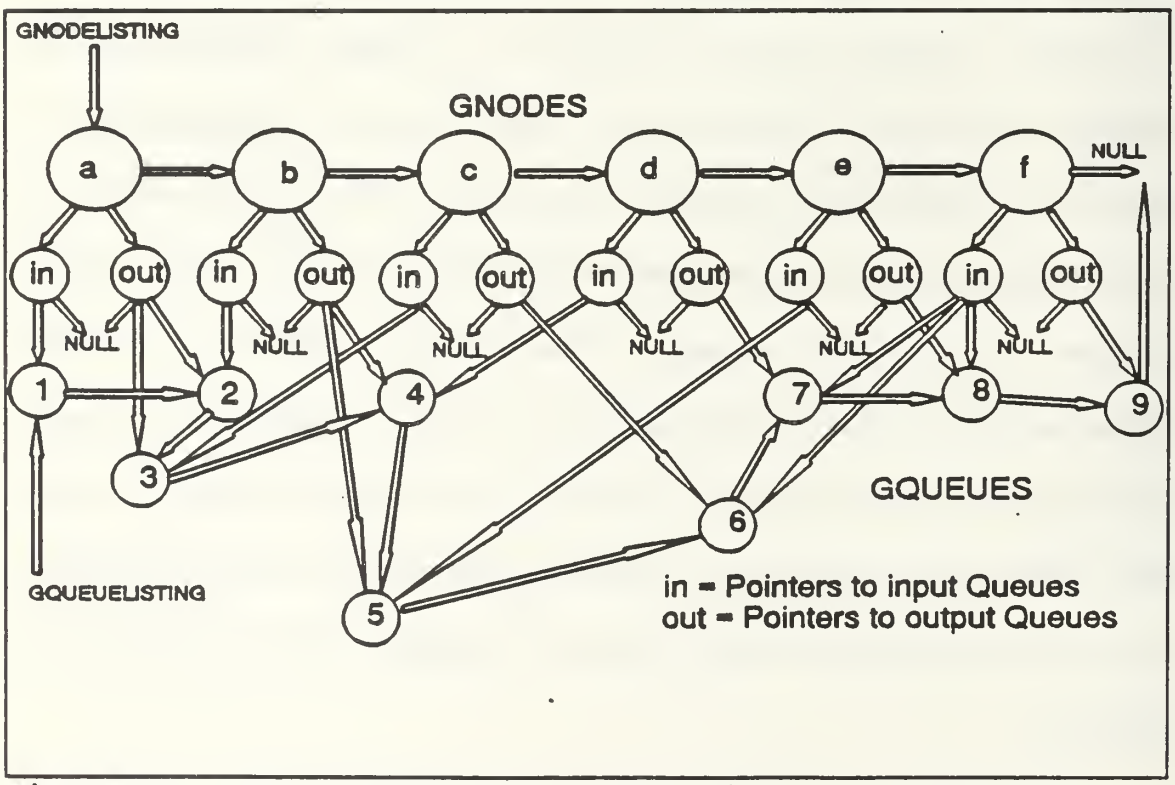

Figure 9: Data Structure Representation of a PGM Graph

Initial design theories revolved around emulating the virtual machine. However, the communication complexities of the AN/UYS-2 virtual machine, dictated that the physical layout of the AN/UYS-2 provide a reasonable method of structuring the software simulator.[1] The software simulator implements a subset of the architecture and instruction set while maintaining operational similarities. The implemented architecture and instruction set, which is loosely based on the *ets++* simulator outlined in ECOS, is demonstrated in Figure 10 [ECOS 89, pp. 4-1 - 4-37]. Each major box in Figure 10 represents a resource element implemented in the simulator. Each set of abbreviations

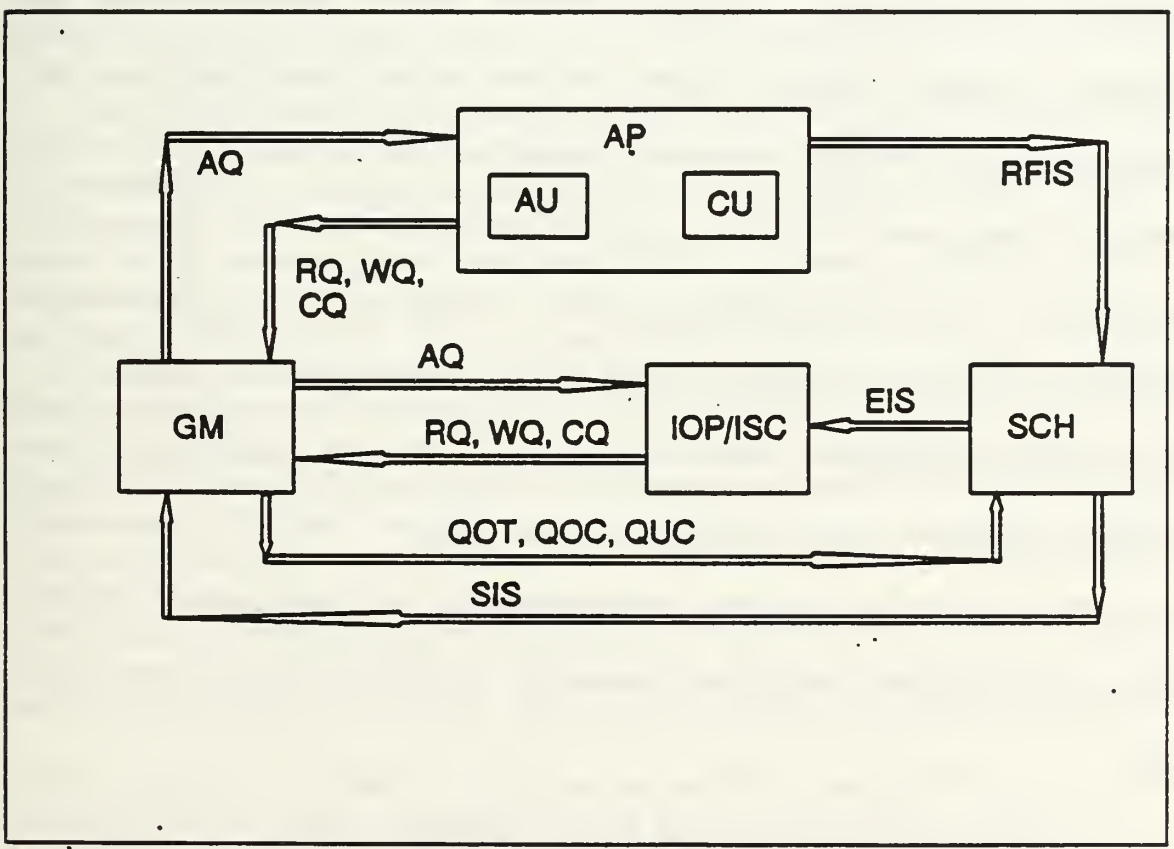

Figure 10: The Simulator Structure

[1] Shown previously in Chapter II, Figure 1.

associated with the arrows represent messages passed between resource elements. The arrows themselves indicate the direction of message flow on either the CBUS or the DTN depending on the message.

1. Communications

Like all distinct entities in the universe and the actual AN/UYS-2, separate objects must communicate with one another. The two main communication nets, DTN and CBUS, are implemented as distinct objects themselves within the simulator. The CBUS is represented as a data structure called *object* and a boolean variable. The boolean variable represents the status of the resource elements to which the CBUS is attempting to communicate. The *object* structure contains information about the following: the object identification number, the time until which the object is busy, the processing status of the object, the transfer status of the object, a pointer to the current *objectnode* being executed by the object, a pointer to the head of the object's input queue, and a pointer to the tail of the object's input queue. The *objectnode* consists of a message instruction, the graphical node identification number associated with that message, the graphical queue identification number associated with the node, the message's origin, the message's destination, and the message's associated location. The DTN is represented by an array of sixteen *object* structures as discussed earlier, each with a boolean variable. Each of the distinct paths established is assumed to be capable of only one transfer at a time.

The main simulator program polls the appropriate CBUS or DTN function *processbus*. *Processbus* determines if this path is waiting for a transfer to be completed

at the desired transfer locations. If it is then no further action is taken this clock cycle. Otherwise, *processbus* conducts the requested message transfer by calling the appropriate functions *commencexfer* and *completexfer*, then processes the next message at the head of its input queue. *Commencexfer* and *completexfer* simulate the required time delay associated with the message transfer and place the message in the destination *object's* queue. The data rate is assumed to be the AN/UYS-2 data maximum of seven mega-words per second. Code for the transfer aspects of the simulator has not been included for brevity.

Since the AN/UYS-2 instruction set consists of over 100 messages, a partial instruction subset consisting of only the most relevant ones was chosen [POPS 90, pp. 1-1 - 10-5]. *Write Queue* (WQ) writes data generated at an AP by a DSP function to the GM responsible for that queue. It incites threshold and capacity crossing information within the GM, spawning *Queue Over Threshold* (QOT), *Queue Over Capacity* (QOC), and *Queue Under Capacity* (QUC) messages [POPS 90, pp. 7-46]. *Queue Over Threshold* is the message by which the GM's inform the SCH that a queue has gone over threshold. It causes the SCH to check all of the queues associated with the node affiliated with the original queue to see if all are over threshold. If they are, it precipitates an *Execute Instruction Stream* (EIS) or *Send Instruction Stream* (SIS) from the SCH to either an IOP or GM [POPS 90, pp. 6-44]. QOC is sent by the GM's to apprise the SCH that a queue has gone over capacity, thereby causing the SCH to suspend further execution of the node feeding the queue until the queue has returned under capacity [POPS 90, pp. 6-43]. QUC is sent by the GM's to apprise the SCH that a queue has proceeded under

capacity. This causes the SCH to continue execution of the nodes feeding this queue [POPS 90, pp. 6-46]. SIS is the message sent by the SCH to a GM initiating the transfer of a node's instruction stream from the GM to the AP indicated in the message. SIS causes the GM to issue an *Accept Instruction Stream* (AIS) to the AP [POPS 90, pp. 7-38]. AIS is the way by which the GM communicates the instruction stream to the AP causing it to load and execute the instruction stream. This execution includes sending the *Read Queue* (RQ) requests back to the GM's for necessary queue data [POPS 90, pp. 9-22]. RQ is sent by the AP to initiate a transfer of data from a queue in the GM to the requesting AP. This message prompts the GM into sending an *Accept Queue* (AQ) instruction to the AP which includes the required queue data [POPS 90, pp. 7-39]. AQ is sent from the GM to the requesting AP in order to transport the required data. This causes the AP to accept and store the specified queue data which was requested with an RQ [POPS 90, pp. 5-131, 9-26]. When the AP is ready to accept the next instruction stream, it sends a *Ready For Instruction Stream* (RFIS) to the SCH. This causes the SCH to attempt to generate a match with a ready node if one is available, if not, then the free AP identification number is added to the free AP list [POPS 90, pp. 6-49]. After the AP has completed breakdown of the last node to execute, it sends a *Consume Queue* (CQ) to the GM that maintains the input queues to that node. This causes the GM's to free the memory that had been used to maintain that data [POPS 90, pp. 7-43]. EIS is sent from the SCH to the IOP's when the queues over threshold belong to a self-regulated or sink node [POPS 90, pp. 10-24].

2. Major Resource Elements

Message passing among the FE's and modular functionality demanded that they be implemented as unique objects. The simulator expects the AN/UYS-2 structure to be in IOP, GM, AP numerical order in a created ASCII file 'EMSPSTRU.' For the SCH and each IOP, GM, or AP specified, the simulator generates an object in memory. Each major element is represented as an object with the exception of the ISC, which is treated as an IOP element. The simulator requests the user to input the type of simulation desired: FCFS or RC, and the last instance of the graph that the user wishes to examine. If RC is selected, the cylinder is created, and dependencies assigned. The expected number of simulated micro-seconds are calculated, and the main simulation begins by repeatedly polling in order: IOP's, CBUS, DTN, GM's, SCH, and AP's. Each loop count represents one simulated micro-second.

Code implementation of the main simulator program and some of the common object code is located in Appendix C. Simulator C++ constructor and destructor code and self explanatory functions are not included for brevity. Constructors generate the initialization specified by the programmer. Destructors delete the pointers designated by the programmer and free the memory storage.

a. The Input/Output Processor

The IOP data structure is represented by another data structure called an *object*, a pointer to a list of nodes assigned to this IOP, and a pointer to the next IOP. Upon initial graph loading, each external input or output node is assigned to an IOP by the graph loading function, *loadgraph*. Each time through the main simulator loop, the

IOP function, *processiop*, is polled. *Processiop* generates the external input queues' Transfers via the DTN are accomplished when the data period specified in the input graph for each external input queue is met as determined by the time of the simulated clock. Once the period is satisfied, a WQ instruction is generated and sent via the DTN to the GM which was designated to retain that queue's information. For each external output node assigned, the IOP acts to execute the node and generate external output queue data when specified by a EIS message from the SCH. No actual implementation of input or output is performed. Code dealing with the IOP implementation including all pertinent functions necessary to access the IOP structures appears in Appendix D.

b. The Global Memories

The simulator GM's do not store the actual queue data involved in node processing. The GM data structure is represented by the data structure *object* and a pointer to the next GM. Each time through the main simulator loop, the GM function, *processgm*, is polled. Through its *object* oriented sub-function *processgmnode*, *processgm* simulates the GM execution as follows. Since queue sizes only change as messages are processed, the simulator GM's perform their queue size determination when WQ or CQ messages are received and generate the appropriate QOC, QUC, or QOT message to the SCH after inserting the appropriate execution delay time. GM messages like RQ or SIS result in redirection and time determination and delay only. Pertinent Implementation code for the GM's appears in Appendix E.

c. *The Scheduler*

The SCH data structure consists of the following: an *object* data structure, a pointer to a list of free AP's, a pointer to a list of ready nodes, a pointer to a list of nodes currently executing on AP's, and a pointer to a list of inhibited nodes. The simulator scheduler emulates the AN/UYS-2 scheduler by acting on the following instructions: RFIS, QOC, QUC, and QOT. Upon receipt of a RFIS message from an AP, the free AP list is updated. A QOC message results in appending the node which supplies the queue associated with the QOC message to the inhibited list. When a QUC message is received, the node in question is removed from the inhibited list. The ready list is not affected since the node is already in this list. The simulator SCH maintains a ready node list which is annotated anytime a QOT message is received from the GM's that results in all queues for the node associated with the message going over threshold. This section of the function *processsch* forms the meat of the simulator SCH. For the FCFS case, the SCH matches the first entry in the free AP list with the first entry in the ready node list. Once the match is made, that ready node is placed in the executing list to inhibit its execution until its current execution has completed. In the RC case, the SCH updates the ready node list and free AP list as above, but before a match is made, the ready node's RC tokens are checked for satisfactory conditions. Only once tokens are satisfied is a match allowed to proceed. Actual code implementation of the SCH appears in Appendix F.

d. The Arithmetic Processors

The AP data structure is represented by three distinct *object* data structures, the number of *set-up* nodes involved, the number of *breakdown* nodes involved, the AP *set-up* status, and the AP *breakdown* status. Three distinct *object* data structures were chosen due to the AP's ability to simultaneously perform set-up, breakdown, and execution on different nodes. The main AP function *processap*, which is polled at every iteration of the main program, performs the *breakdown*, *execution*, and *set-up* status checks and forwards the node message information between stages. The AP's simulate the actual node *setup*, *breakdown*, and *execution* by entering delay cycles a corresponding to that nodes execution data loaded at simulator run-time. Upon the transfer of information between the *set-up* and *execute* stages, which implies that a new node is now being executed, the AP issues a RFIS message to the SCH. Through its sub-function *processapnode*, *processap* performs the message actions and redirection required. The AP's act as receptors for the AIS message sent by the GM's. Upon receipt of the AIS message, the *set-up* stage of the AP determines the number of input queues required and issues that many RQ instructions to the GM's associated with the AIS message. Upon receipt of an AQ message from the GM's, the AP *set-up* stage updates its delay time and waits for the last AQ message that it is expecting for that node. After receiving the last AQ message, it executes a transfer of information to the *execute* stage if the *execute* stage is not busy. Otherwise it waits until it is able to perform the transfer. When the *breakdown* stage receives the information from the *execute* stage, it generates a WQ instruction for every output queue associated with the node that was executed and

generates a CQ instruction for every input queue associated with that node. Implementation code for the AP's appears in Appendix G.

B. THE LANGUAGE - C++

During the 1980's, the C programming language matured through the addition of classes, type checking and conversions, virtual functions, and operator overloading until a new language, C++, resulted [STROUSTRUP 86, pp. 1-12, POHL 90, pp. 5]. "Since its conception, C++ has been evolving to meet the needs of its users." [STROUSTRUP 90, pp. 3] C++ has endured much use on large software projects. Its stability, compatibility, space-efficiency, and run-time features have been strongly documented by Ellis, Stroustrup, and Pohl [POHL 89, pp. 1-25].

Since the AN/UYS-2 and C++ 2.0 are products of AT&T Bell Laboratories, C++ was considered a forerunner in the choice of programming languages. C++ was chosen to match the modularity of the AN/UYS-2. The reasons for which C++ 2.0 was chosen over Ada include:

- Object Oriented Programming
- Encapsulation
- Inheritance.

Object oriented programming allows *abstract data types* to be constructed so as to allow the programmer to model the problem domain within a *class* structure. A *class* can be thought of as an extension of the idea *struct* from traditional C. The AN/UYS-2's distinctive modularity of components lends itself precisely to programming with objects.

Encapsulation refers to C++'s ability to completely self-enclose both the internal implementation details of the type and functions that act on objects of that type from the external user [POHL 89, pp. 2,211]. Encapsulation ensures that unqualified simulator personnel will not be able to access or blindly modify the simulator objects.

Stroustrup defines inheritance as the mechanism of deriving a new class from an existing base class [STROUSTRUP 90, pp. 2]. Due to the hierarchical structure of the AN/UYS-2, inheritance guarantees that key conceptual ideas remain intact from *object-to-object.*

C. USER INTERFACE

To execute the simulator the user simply invokes the executable version and follows the on-screen prompts. The simulator collects data about the individual graph nodes. The time that sink nodes are sent to the IOP's are output to a file 'results'. The time that an instance is placed onto the ready list and the time at which an instance is removed from the ready list are used to calculate the time an instance spends on the ready list. This time gives a good indication of the delay time involved in scheduling a node. The time that a node spends between breakdown completion and its successor node being placed on the ready list is calculated to provide an indication of communication delay. Similarly, the time between a node being removed from the ready list until it arrives at an AP also provides information about communication delay. The time which the AU portion of the AP spends not busy is calculated to provide details about set-up and breakdown delay times. This information is provided to the user on a screen display

upon completion of the simulation for the node instances specified by the user. Currently only a span of twenty node instances can be examined per simulation run.

D. LIMITATIONS OF THE SIMULATOR

Like any full scale simulator, this AN/UYS-2 simulator is not intended to be of all inclusive design. The simulator IOP assumes that an unlimited buffer is available for incoming data. Execution of self-regulated nodes in IOP's is not implemented. The ISC's are simplified to IOP's for the simulation. The initial assignment of queues to GM's is assumed to be completely known upon initialization. All input queues associated with a node are assigned to the same GM by the simulator. Unlike the actual AN/UYS-2, the simulator is severely restricted by the amount of memory available on which to run it. While the simulator is written to be ported between any machine capable of C++ version 2.0, large-grain data-flow graphs quickly fill available memory and disk space. The time required to perform each simulation discussed in Chapter V exceeds four hours on a 33-mega-hertz 80486 (16.11 Million Instructions Per Second).

V. PERFORMANCE EVALUATION

As mentioned previously, a digital signal processor's performance can be measured by throughput. Unfortunately, the AN/UYS-2's throughput is not based solely on its scheduling algorithm. The number of nodes and queues in the graph and their sizes contribute to the overhead involved. For this reason several types of application graphs were chosen for analysis of the AN/UYS-2 performance as it pertains to the scheduling algorithms. All of the results obtained are based on varying the input data rate.

A. CORRELATOR APPLICATION

The initial correlator application was chosen to provide an actual *on-hands* non-uniform ECOS graph as outlined by ECOS [ECOS 90, pp. H-CG-5 - H-CG-27].

1. Description

The graphical diagram for this graph appears in Figure 11. Again, the circles indicate primitive nodes to be executed and the arrows represent the queues that manipulate the data required by the nodes. "T" represents the threshold value required for that queue before scheduling of the subsequent node can be arranged. "R" represents the amount that is read by the subsequent node on execution set-up. "C" represents the amount that is consumed on subsequent node breakdown. "P" represents the production amount from the previous node. Actual execution times for the primitives listed beside the nodes were calculated by use of the Graph Primitives Library and are included with

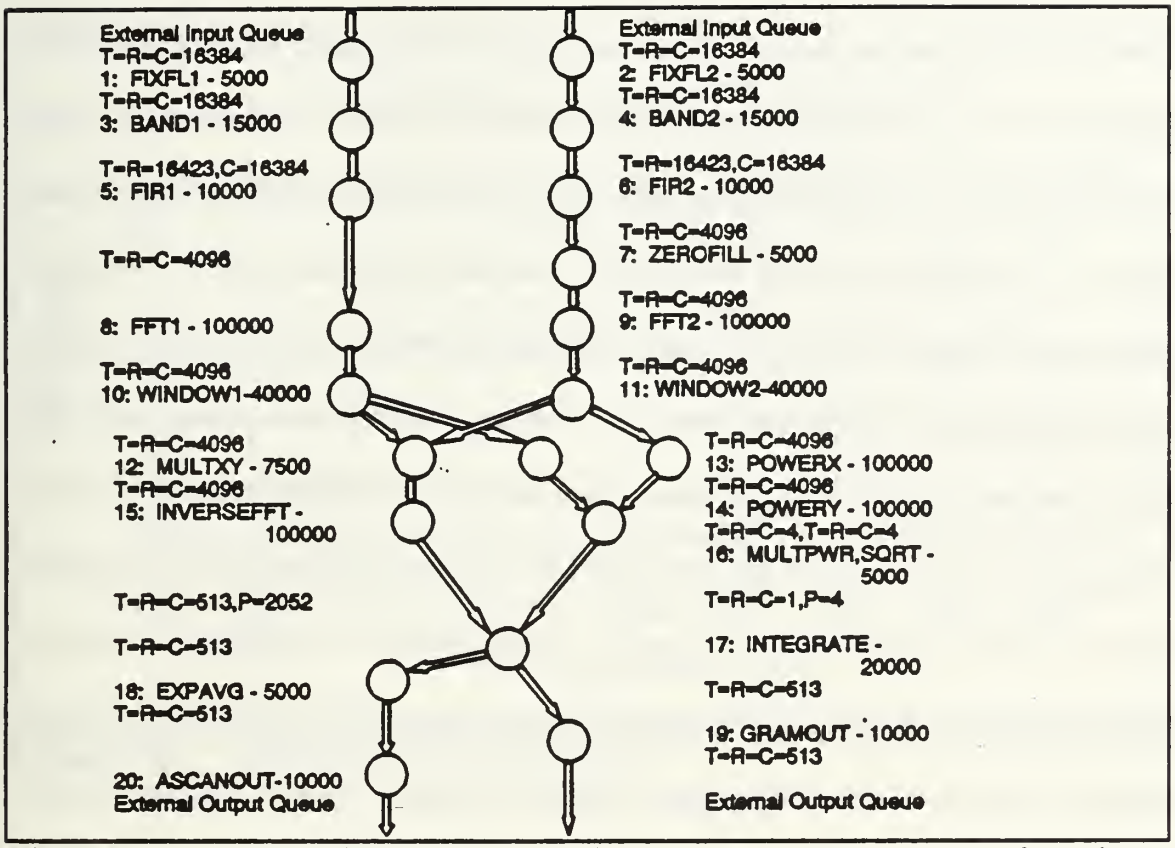

Figure 11: Graphical Description of Correlator Application

the primitive name. The graph would be input to the simulator after performing the manual conversion to the standard numerical format described in Chapter IV.

2. Output and Interpretation

The points obtained for the graphs plotted in the case of the correlator graph were taken at five percent intervals except in the region of close similarity where the interval was one percent. The *normalized input data interval* refers to the theoretical maximum throughput rate for the application assuming no internal delays. As discussed in Chapter III, the theoretical maximum throughput rate is based on the total execution time for the graph divided by the number of AP's in the configuration. Therefore, the

normalized input data interval axis is based on interval times greater than the theoretical maximum of 1.0. *Normalized mean*, which refers to the mean time between output arrivals divided by the theoretical input data interval, results are shown in Figure 12 and Figure 13. The closer the value to uniformity, the better the performance. While not discernible in Figure 12, Figure 13 clearly indicates that the RC algorithm reaches unity *five* percent before the FCFS algorithm. At all times the RC curve remains below the FCFS curve on the graph. The correlator graph *normalized* standard deviation is shown in Figure 14. The *normalized* standard deviation curve indicates that the RC algorithm provides a more uniform output than does the FCFS algorithm throughout the range of input data intervals. Figure 15 demonstrates this output arrival by plotting the *normalized completion time* for the first thirty graph instances observed. Due to the dependencies inserted by the RC algorithm, the AP efficiency is lower for the RC case than for the FCFS case until uniformity in output is obtained as shown in Figure 16. This result is caused by the dependencies inhibiting the earlier nodes in the graph from executing until they are satisfied.

While the AP efficiency is slightly lower for the RC approach, the lower normalized mean and standard deviation results indicate a slight improvement by use of the RC algorithm over the current scheduling technique. The output times associated with the thirtieth observed graph instance also bears out this analysis.

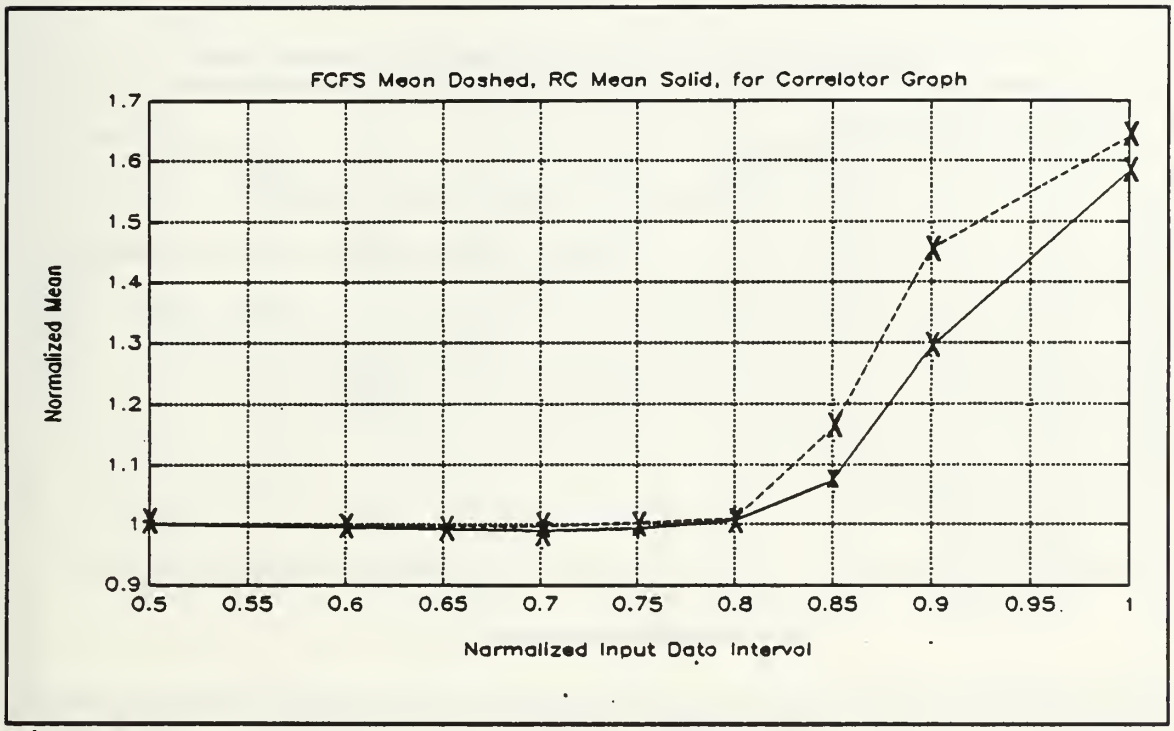

Figure 12: Correlator Graph Mean

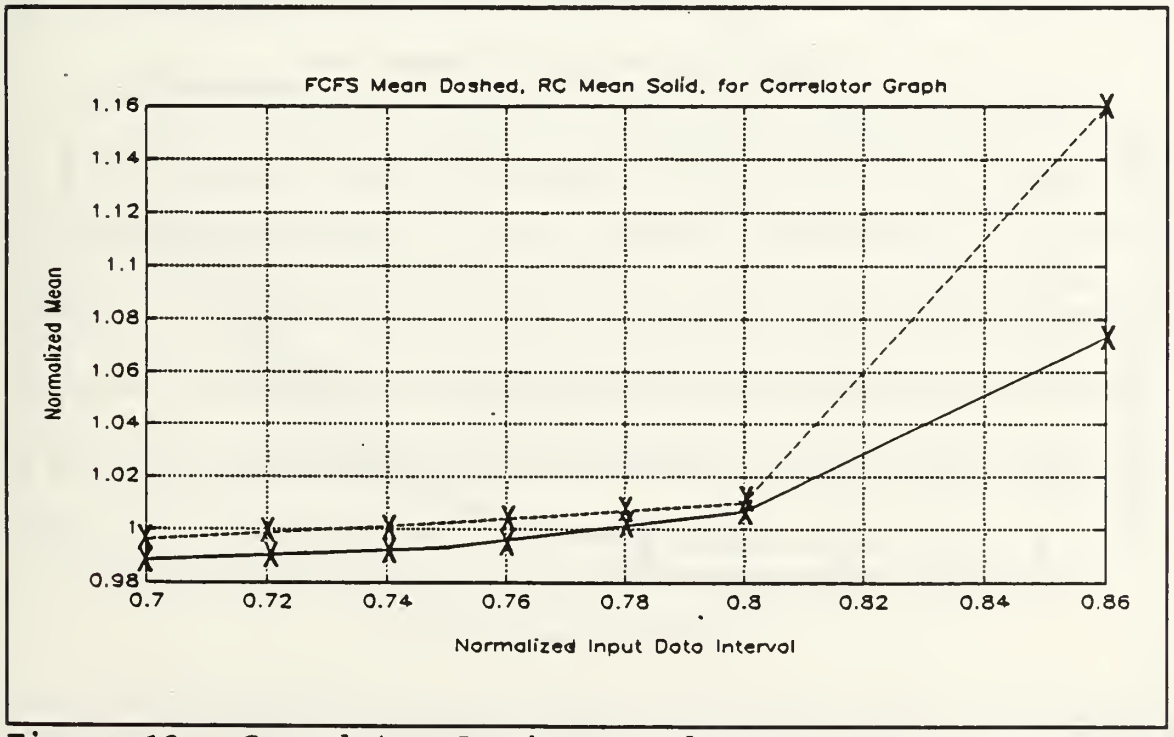

Figure 13: Correlator Graph Mean Blow Up

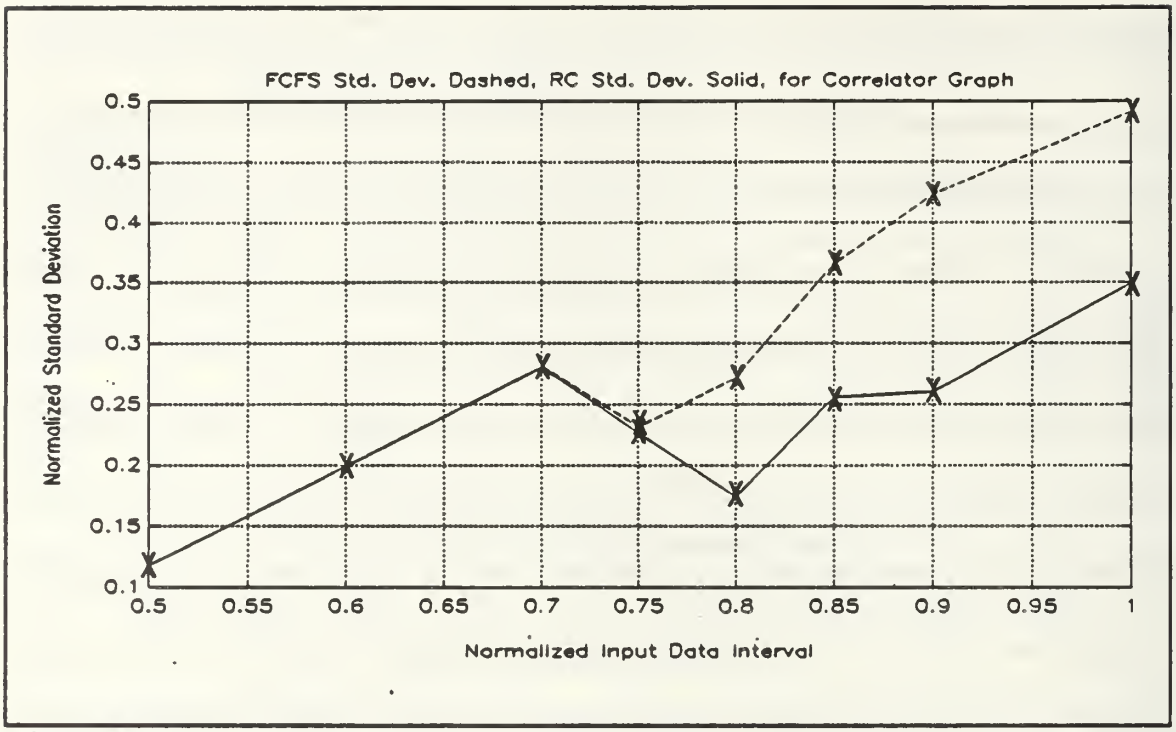

Figure 14: Correlator Graph Standard Deviation

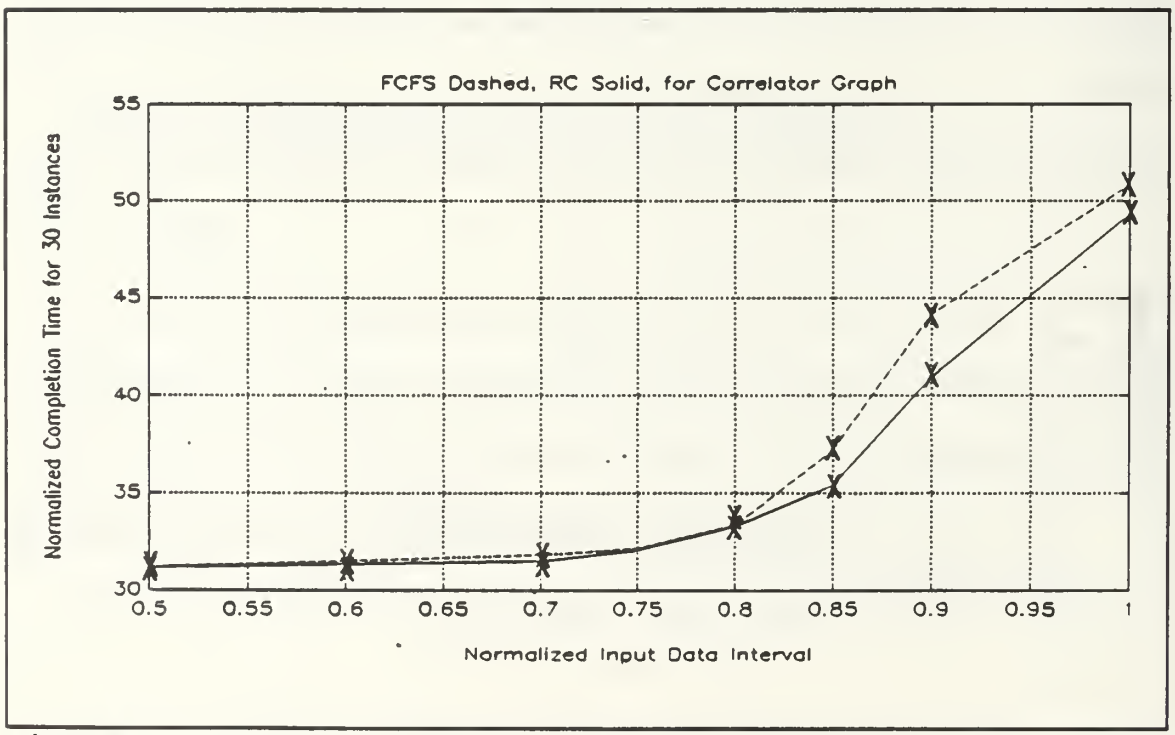

Figure 15: Correlator Graph Normalized Output Completion Time

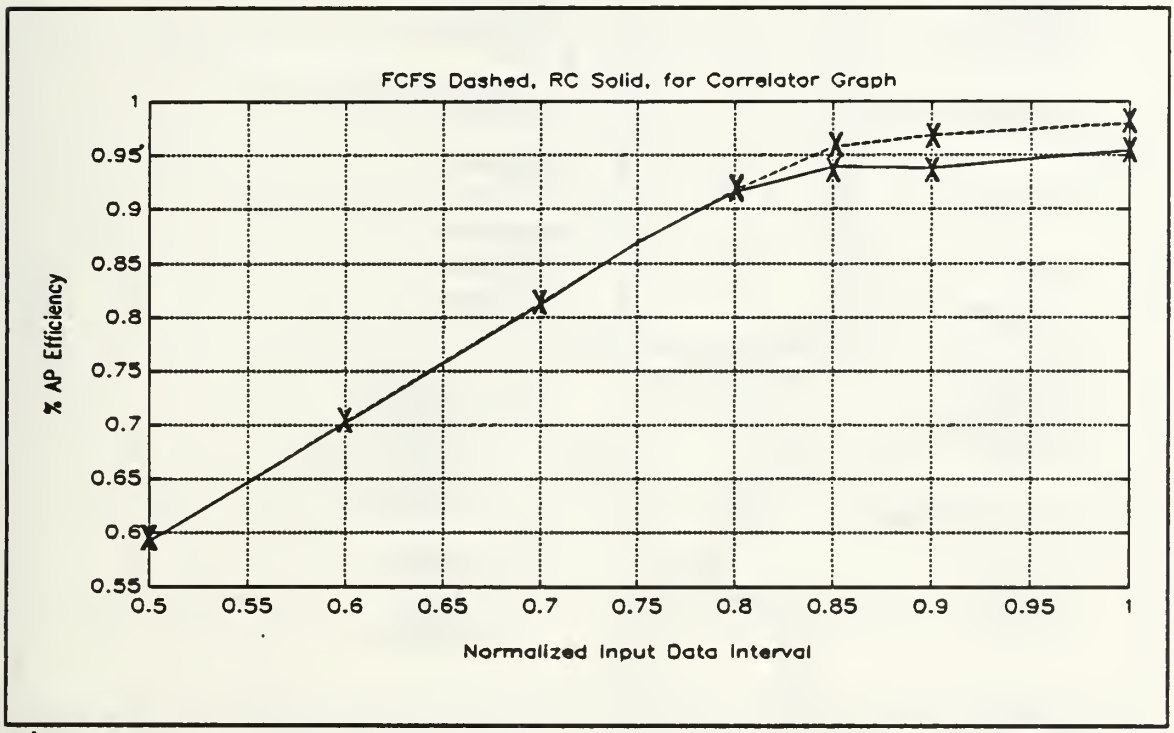

Figure 16: Correlator Graph AP Efficiency

B. CORRELATOR WITH UNIFORM NODE SIZES

1. Description

To examine the effect of varying execution times on the nodes in the application, the same graphical structure was maintained, but the execution times of the nodes were changed to be uniform as shown in Figure 17. The uniform size of 36000 micro-seconds per node was selected based on maintaining the same overall execution time of the graph. It was assumed that primitives could be restructured to meet this size without major revisions to queue sizes required.

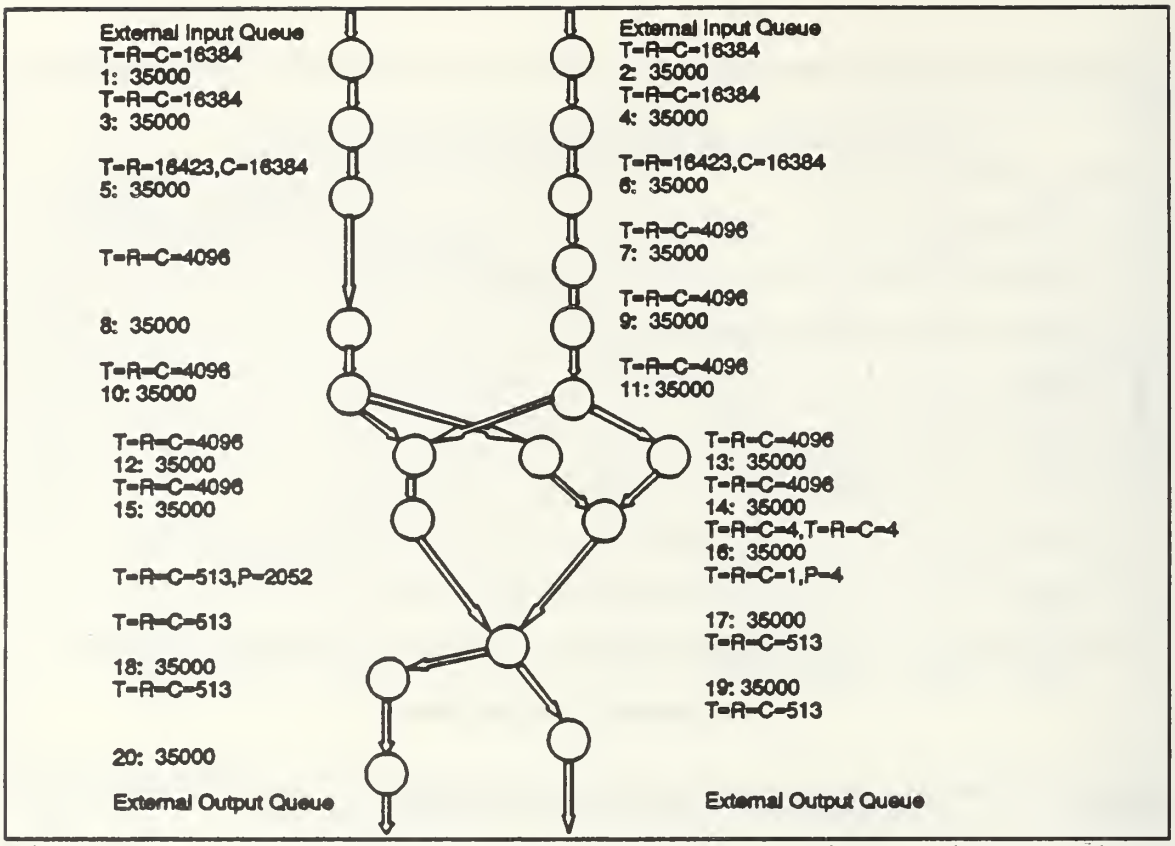

Figure 17: Correlator Graph Structure with Uniform Sized Nodes

2. Output and Interpretation

The *normalized means* for the correlator graph structure with equal execution time nodes is shown in Figure 18 and Figure 19. It can be seen that as long as the input data rate is not being met, the RC algorithm performs better than the FCFS algorithm. But, both the RC and FCFS algorithms reach the ability to meet the input rate at the same time. The *normalized* standard deviation curve is shown in Figure 20. The RC standard deviation never rises above 1.0. Therefore, the RC algorithm produces a more uniform output than does that of the FCFS algorithm. Figure 21 demonstrates that as long as the input data rate is not being met, the RC algorithm completes thirty graph instances before

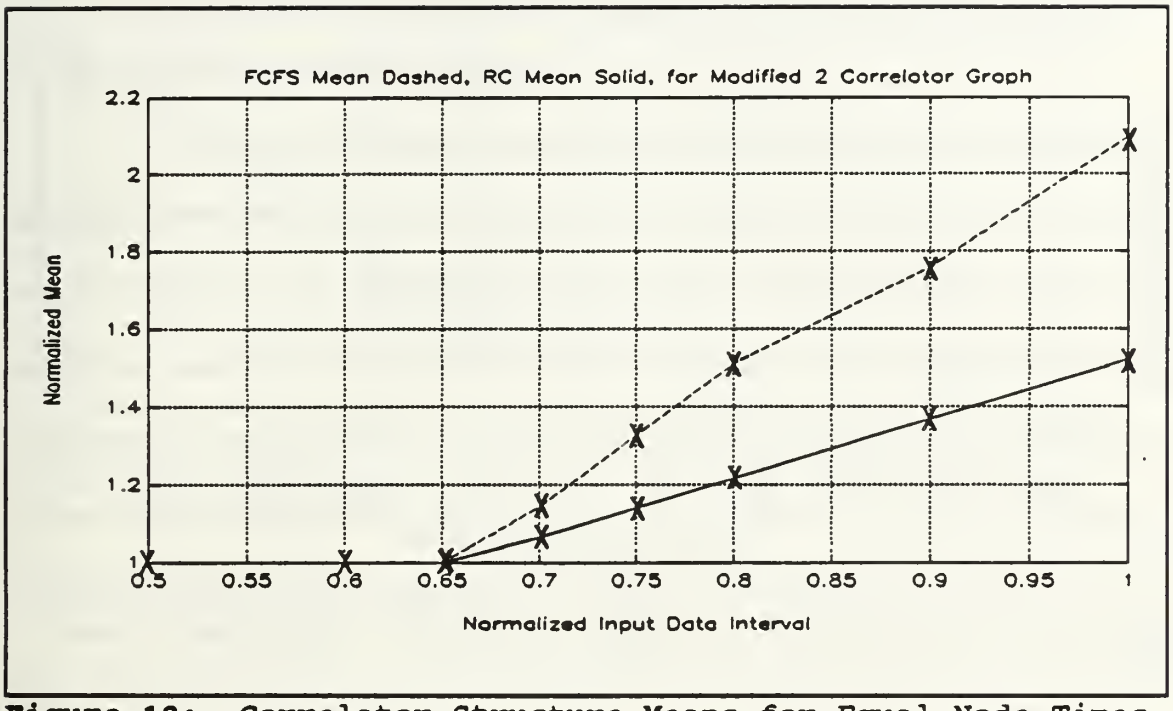

Figure 18: Correlator Structure Means for Equal Node Times

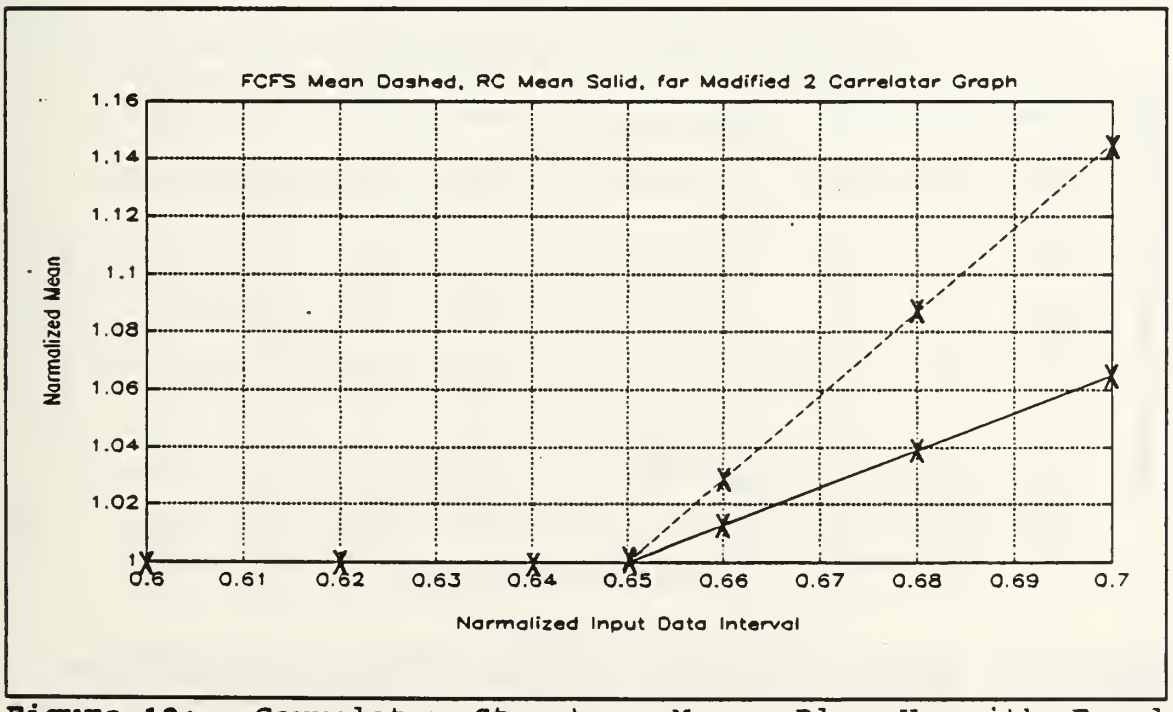

Figure 19: Correlator Structure Means Blow Up with Equal Times

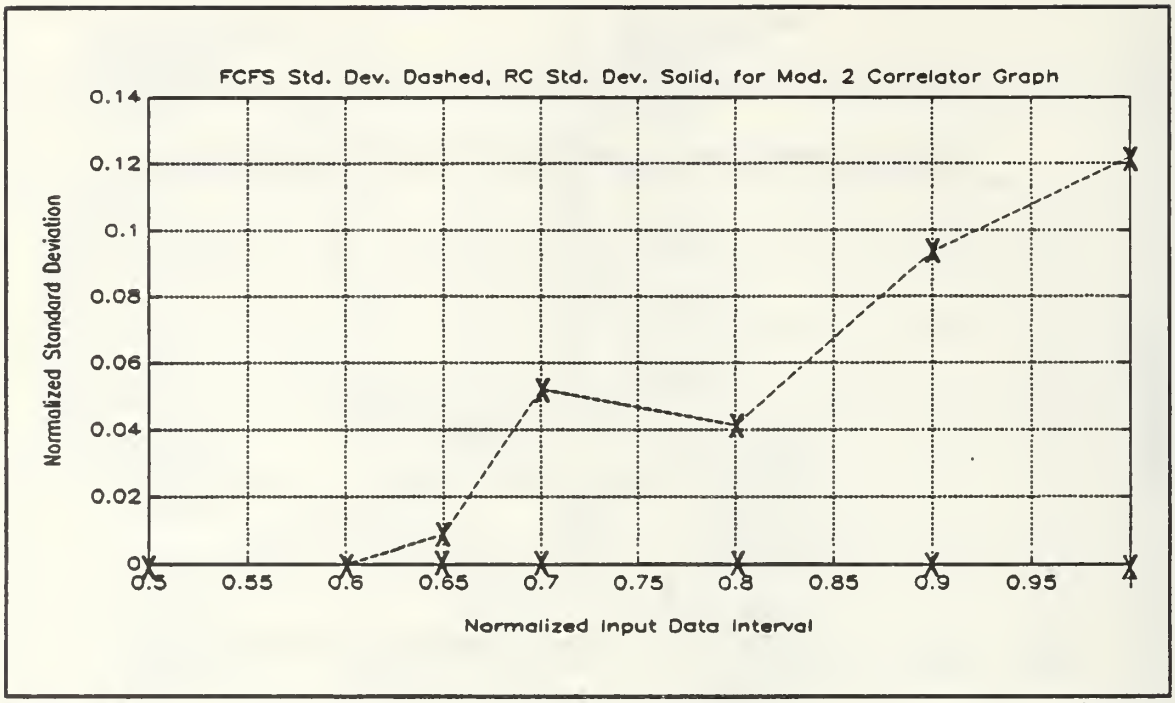

Figure 20: Correlator Structure Deviation with Equal Times

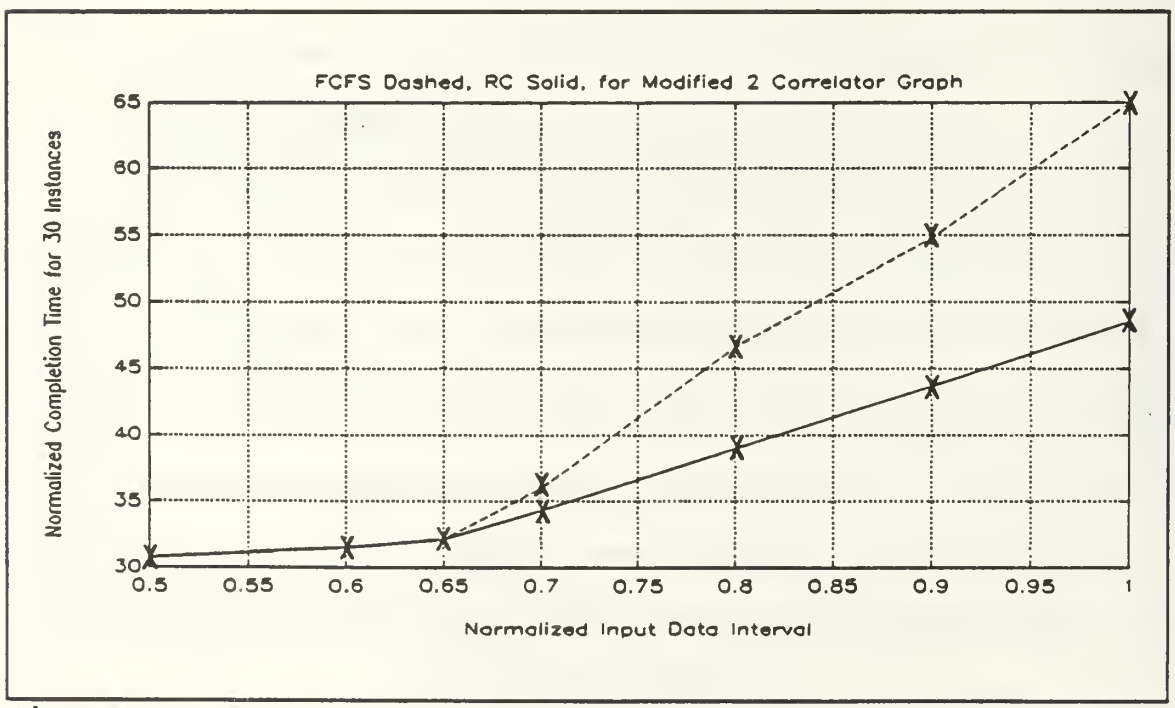

Figure 21: Equal Correlator Structure Output Arrival Times

the FCFS algorithm. The percent AP efficiency plotted in Figure 22 is the same for both the RC and FCFS scheduling algorithms.

Since the AP efficiency is the same for both the RC and FCFS cases, the dependencies inserted by the RC algorithm do not result in slowing the AP utilization rate. While this is an improvement over the first correlator graph examined, the normalized means for the correlator structure with equal size nodes reach the 1.0 value at precisely the same instance, 65%, tending to indicate no improvement. However, the normalized standard deviation for the RC approach never varies from 0.0. This indicates a perfectly uniform output regardless of the load level, including a five percent band after both the RC and FCFS normalized means reach 1.0. Again, this indicates a slight improvement for the RC case.

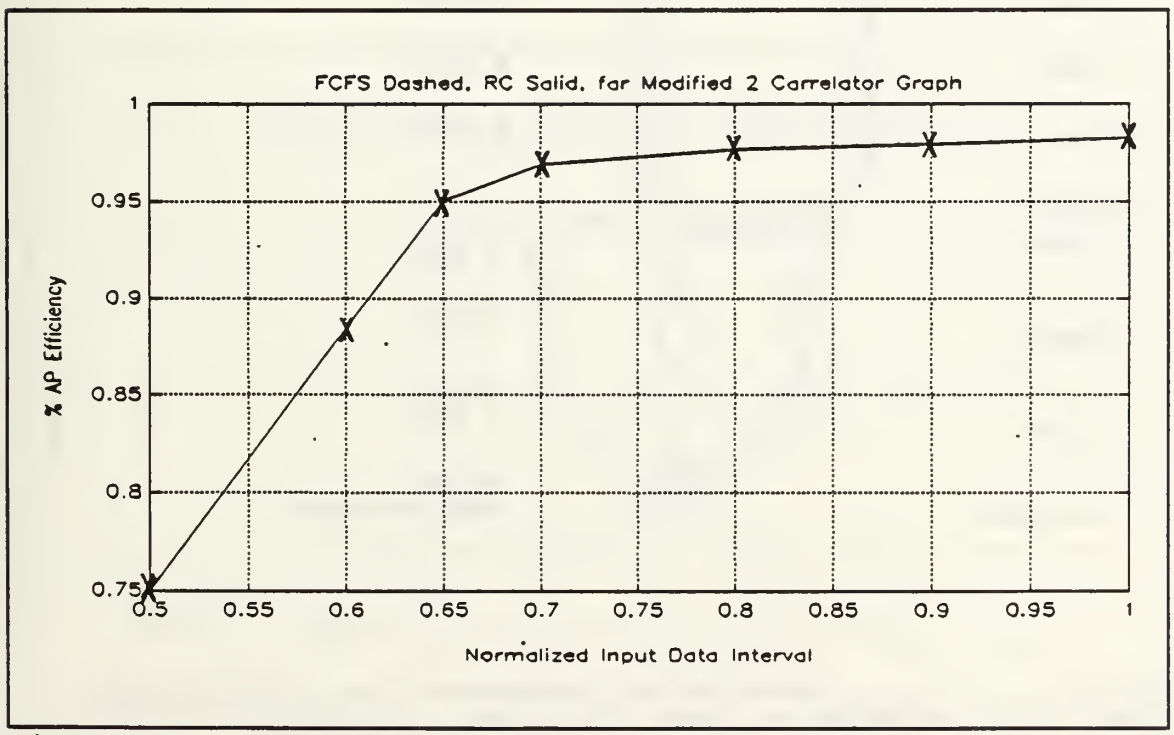

Figure 22: AP Efficiency for Correlator Structure Equal Times

C. CORRELATOR WITH CHAINED NODES

In order to examine the effect of the graph structure as compared to the uniform sizes of the node, the original correlator graph was *chained* to uniform node sizes. This *chaining* of nodes resulted in a different graph structure with different sized nodes. All of the queue information is assumed to remain consistent.

1. Description

The graphical representation of the correlator graph is shown in Figure 23. The execution times for the nodes are documented beside them. This graph was derived

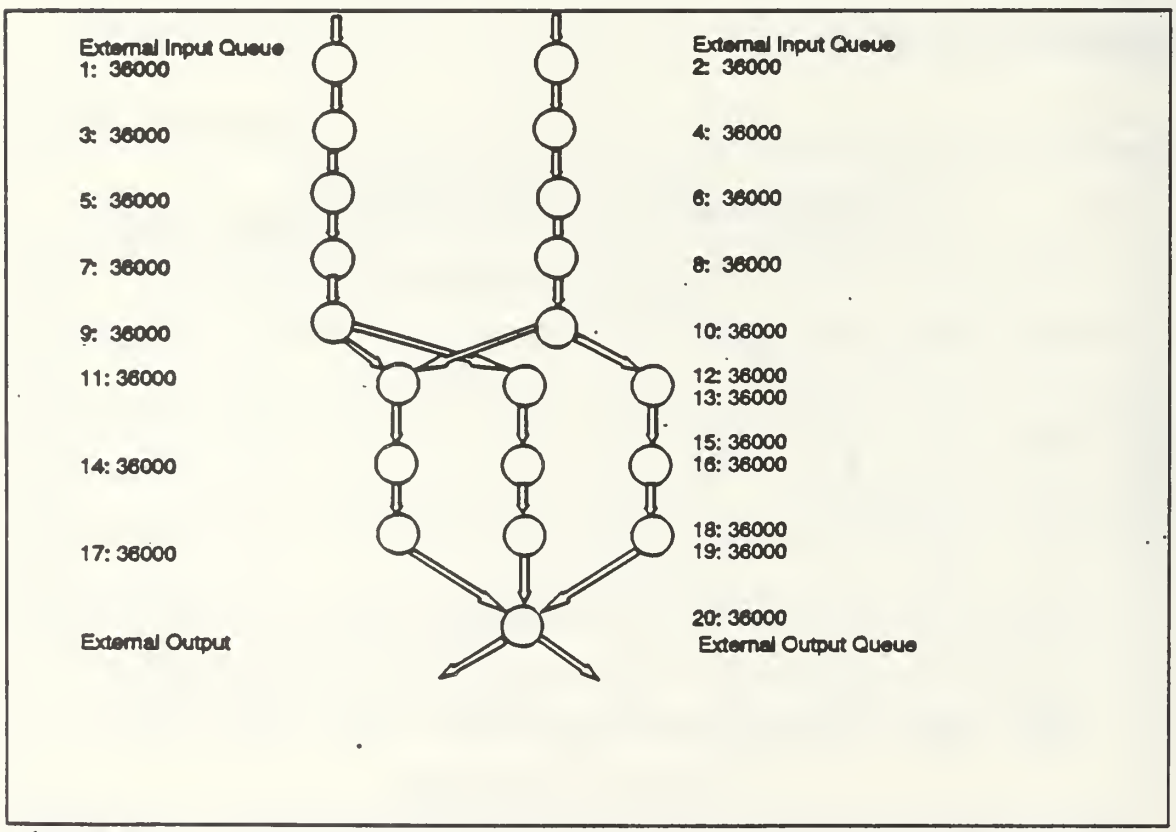

Figure 23: Correlator Graph Structure with Chained Nodes

from the basic correlator graph by again minimizing the variation in total execution time. However, this graph possesses a different graphical structure from the correlator graph with equal execution times because in this graph the actual individual primitives have been chained and segregated as best that could be determined.

2. Output and Interpretation

The normalized means are shown in Figure 24 and Figure 25. Both the RC and FCFS algorithms possess the same normalized means even before the input data rate is met. Figure 26 shows the *normalized* standard deviation. While the RC algorithm maintains at least as well a *normalized* standard deviation as the FCFS algorithm, the relative closeness of the numbers precludes any explicit determination. The normalized observed instance completion times for thirty graph instances are shown in Figure 27.

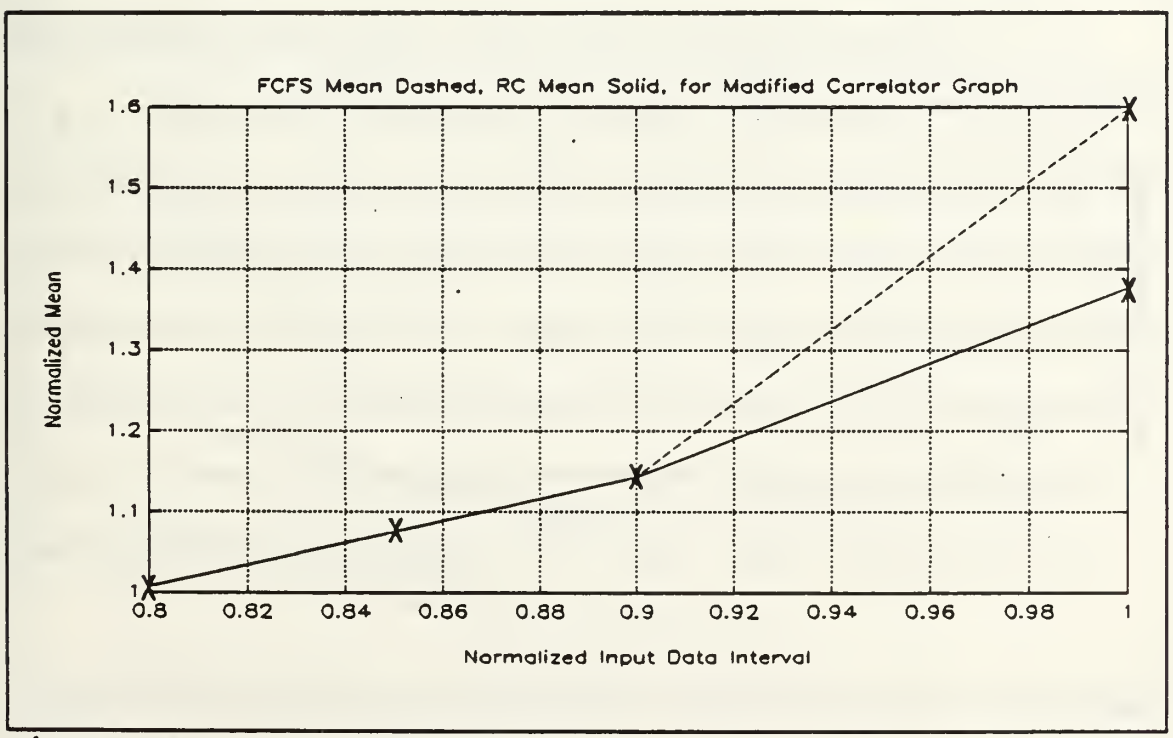

Figure 24: Correlator Graph Structure Mean with Chained Nodes

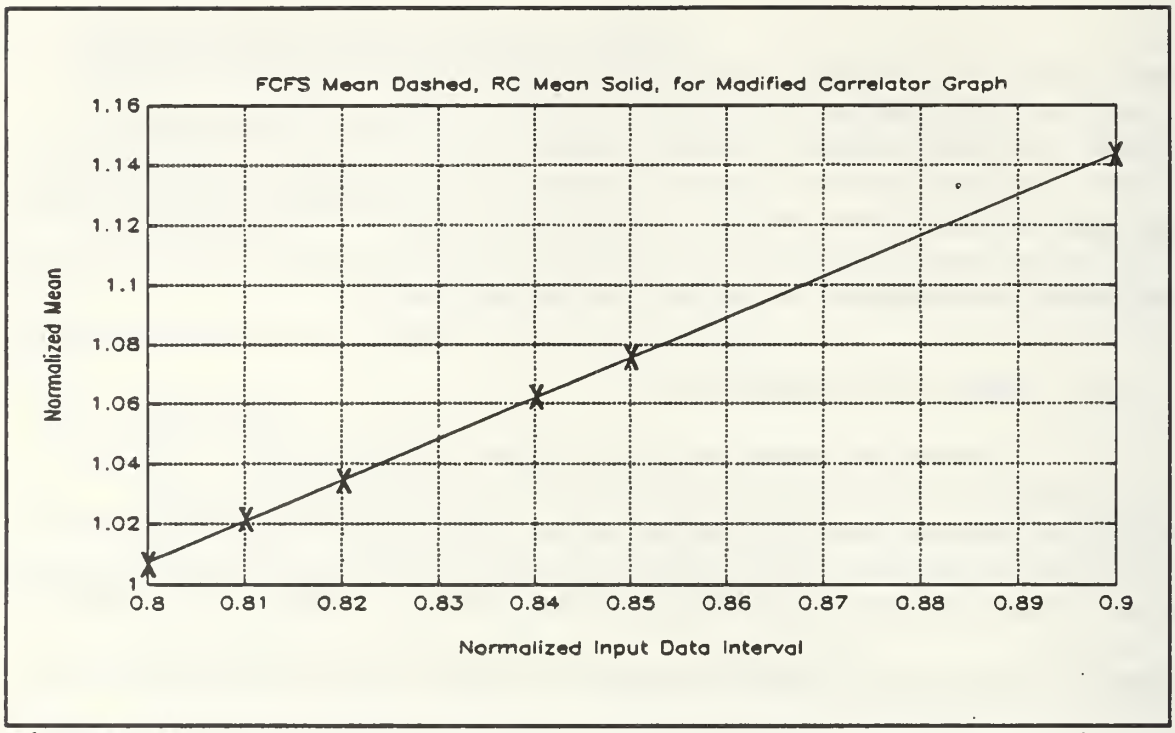

Figure 25: Correlator Graph Structure Mean Blow Up Chained

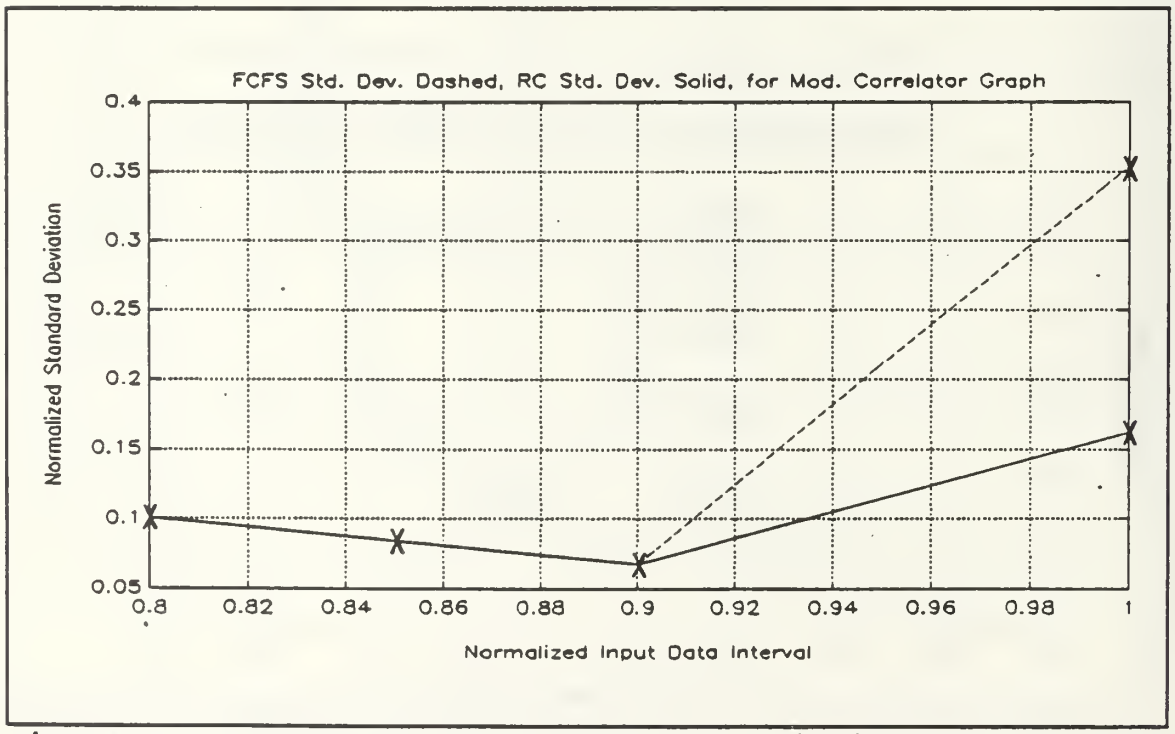

Figure 26: Correlator Graph Structure Deviation Chained Nodes

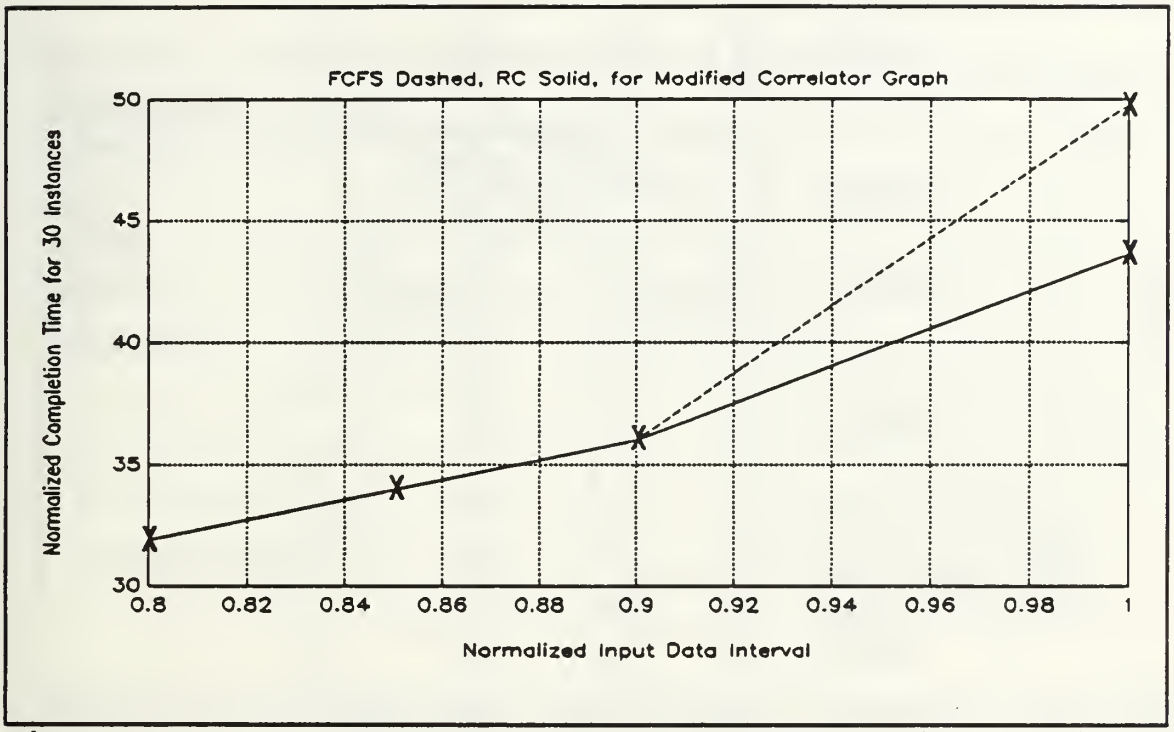

Figure 27: Correlator Graph Structure Output Times Chained

Figure 28 demonstrates a slight AP efficiency difference between RC and FCFS algorithms.

Again, once the input data rate is close to being met, the RC and FCFS algorithms perform the same. Yet, for data rates which are not met, the RC algorithm completes the instances sooner than does the FCFS algorithm with a lower normalized standard deviation. But, unlike the correlator graph or the correlator graph with equal nodes, there is no five percent improvement overlap seen in the mean and standard deviation. Therefore, it appears that once the input data rate is met, chaining performs equally as well as the RC approach.

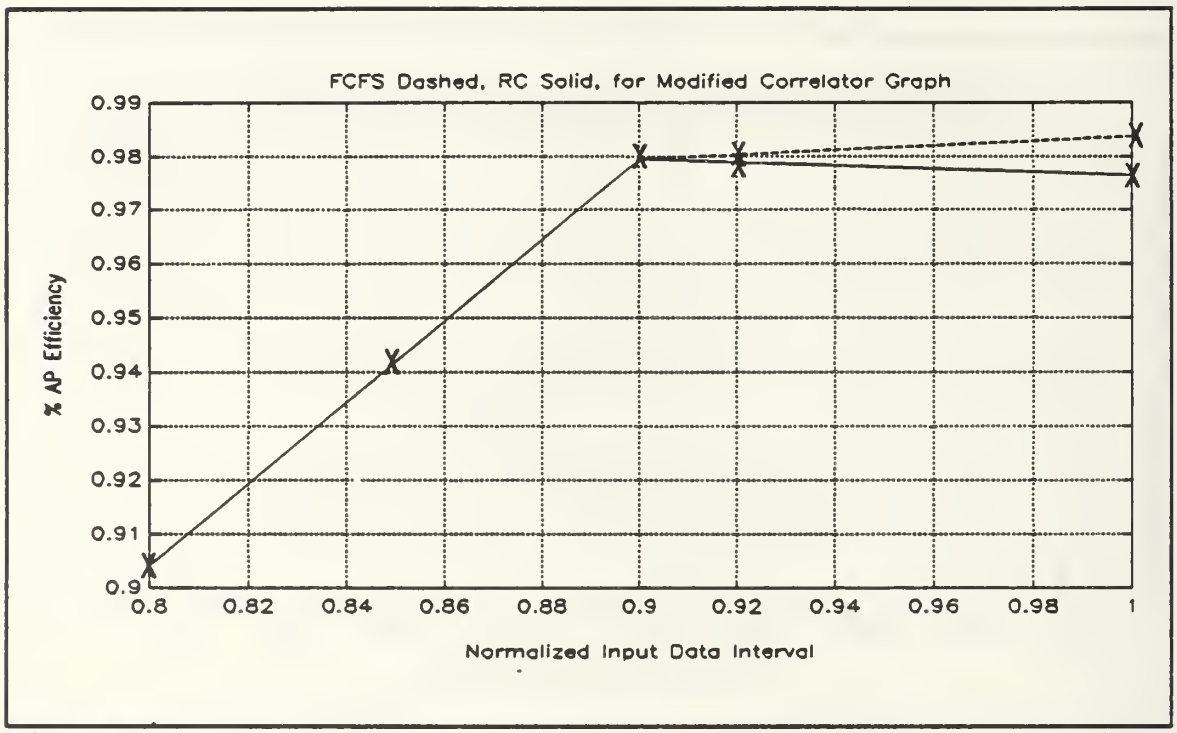

Figure 28: Correlator Graph Structure AP Efficiency Chained

D. FFT APPLICATION

A Fast Fourier Transform (FFT) Application was chosen to examine the effects of the RC scheduling algorithm on a large scale communication intensive interconnected application. A communication intensive application is characterized by large queues resulting in high communication overheads on the DTN and CBUS for message passing. This typically results in low overall AP efficiency.

1. Description

The data-flow graph for a two dimensional (2-D) FFT can be represented in terms of that of a one dimensional (1-D) FFT. This application assumes a 256 point vector of inputs. The 1-D FFT shown in Figure 29 can be calculated in *log 256* (8)

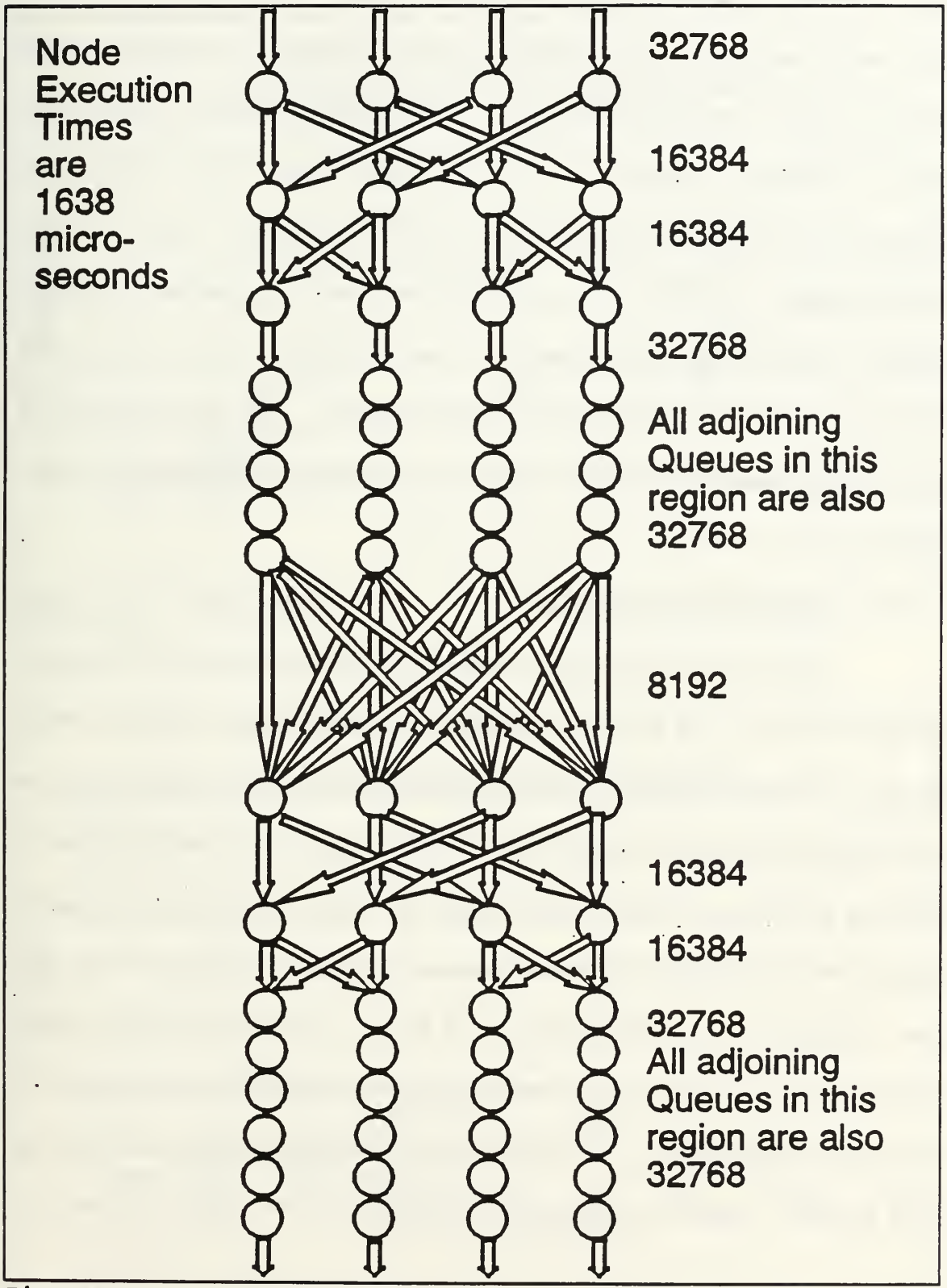

Figure 29: FFT Graph Description

stages of *butterfly* operations with 128 *butterflies* per stage. Each stage can be divided into *p* parallel tasks, with *256/2p* butterflies per task. As the tasks in stage *i* finish, they send their outputs to the tasks in stage *(i+1)*. The data-flow graph for a 2-D FFT uses *2log 256* (16) stages to transform a *256 x 256* matrix of inputs. *256* 1-D FFT's are computed for rows followed by another *256* 1-D FFT's for columns. Tasks in the first 8 stages perform 1-D FFT's on all *256* rows with each task performing $256^2/2p$ butterflies. Tasks in stage *log 256* send data to tasks in stage (8 + 1) in such a way that the second set of 8 stages performs 256 column transforms. The numbers beside the queues represent *queue over threshold*, *production*, and *consume* values in micro-seconds. [SHUKLA 90, pp. 48-51]

2. Output and Interpretation

The *normalized FFT means* are shown in Figure 30 and Figure 31. Here also, the input data rate is met for the RC algorithm five percent before that of the FCFS algorithm. The input data rate is not met until further down in the percentage range due to the high communication overhead involved with this graph. The *normalized standard deviations* are shown in Figure 32 and Figure 33. Again, clearly the RC standard deviation out performs the FCFS standard deviation throughout the spectrum of input data rates. Also note the consistency that exists in the RC *normalized standard deviation* across the spectrum. Figure 34 documents the observed *normalized completion times* for the first thirty graph instances. The RC algorithm generates the results quicker than the FCFS algorithm. Figure 35 demonstrates the differences in AP efficiency for the FFT

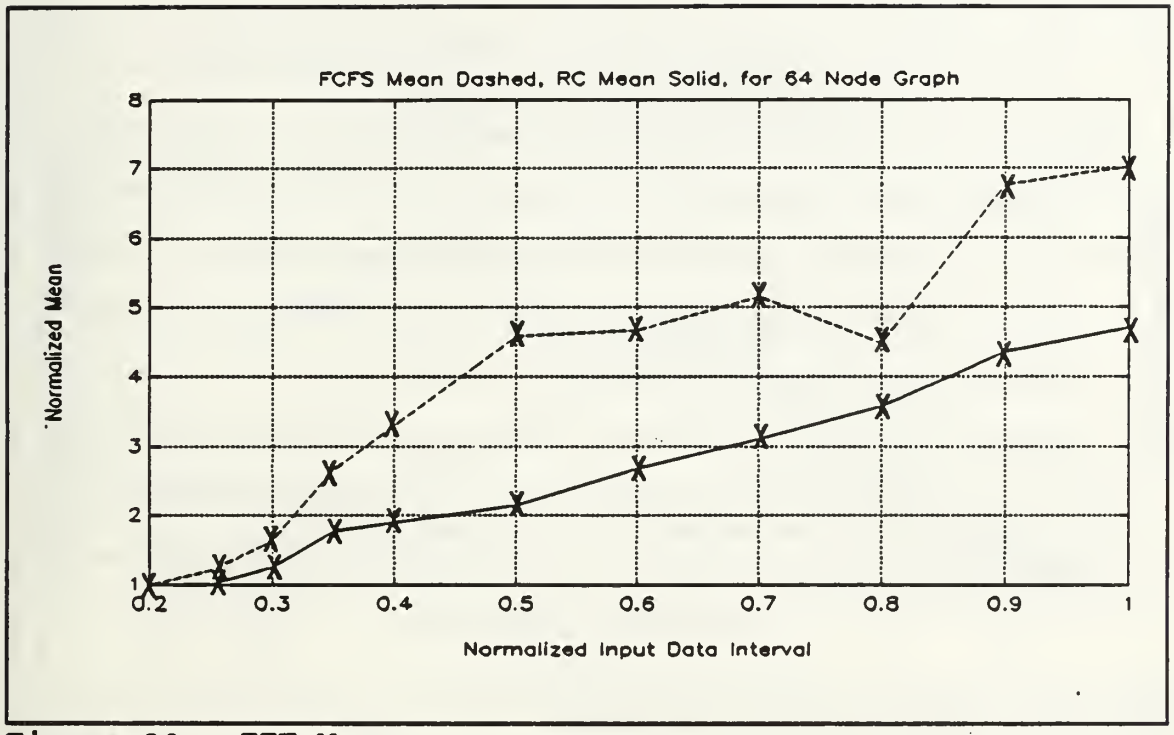

Figure 30: FFT Mean

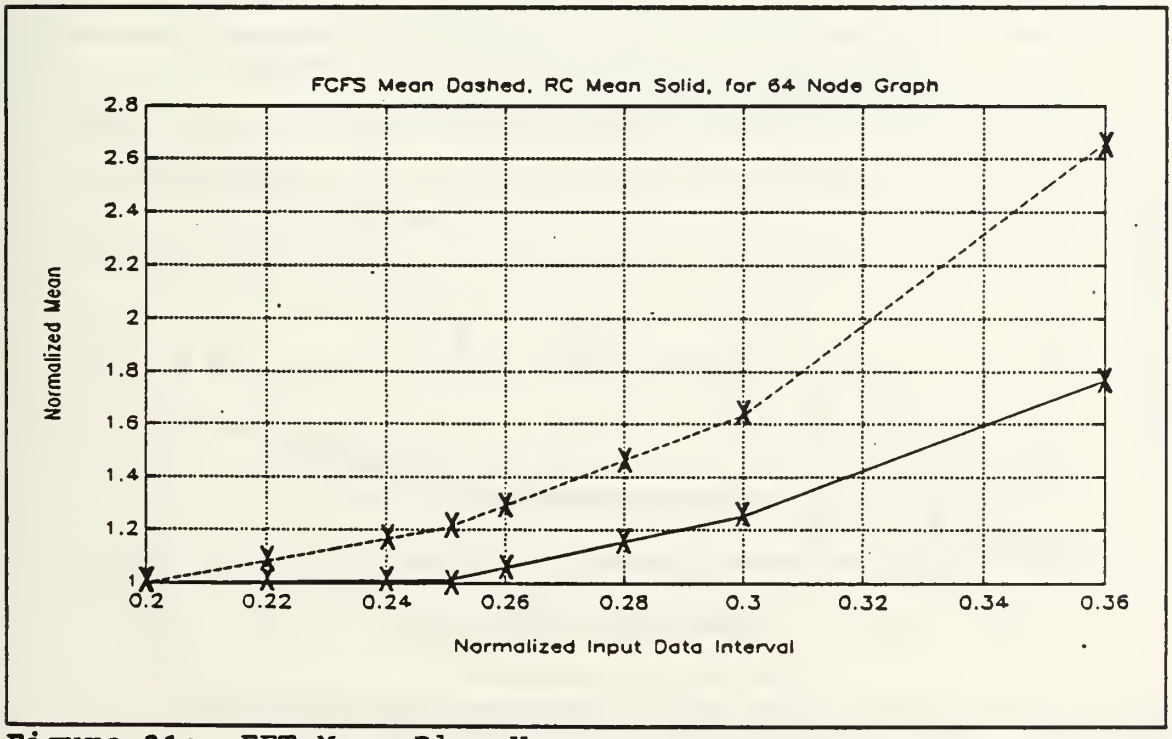

Figure 31: FFT Mean Blow Up

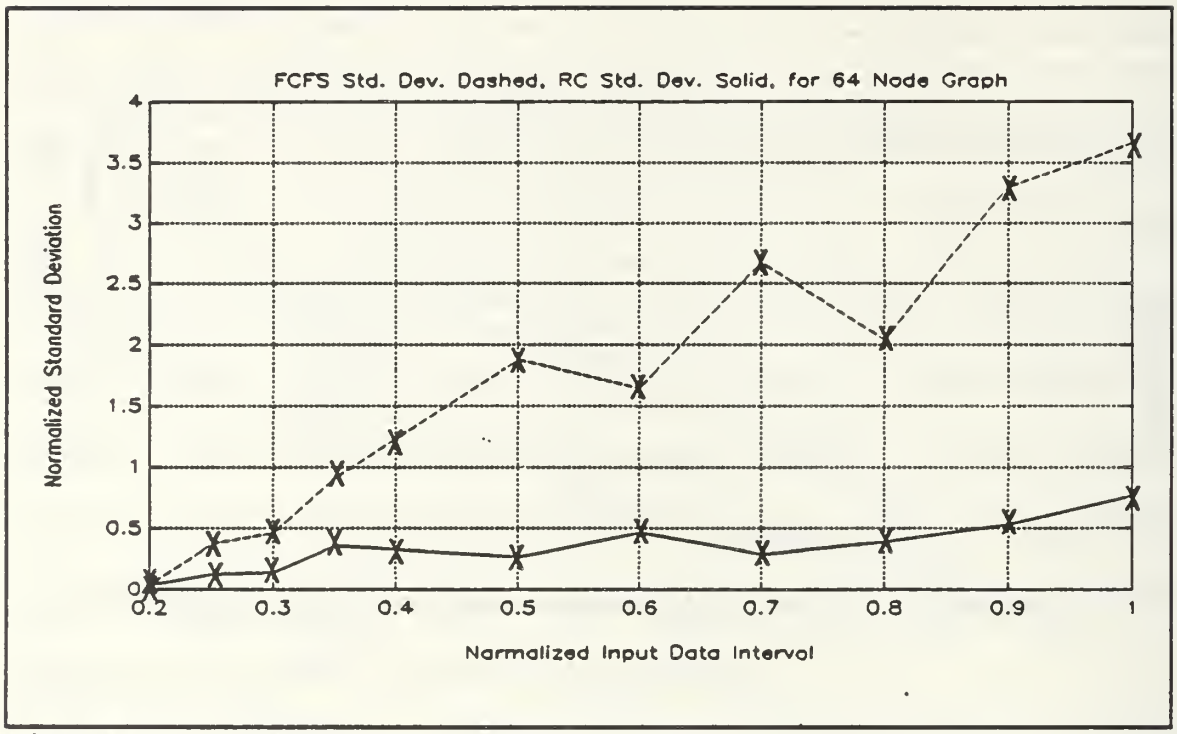

Figure 32: FFT Standard Deviation

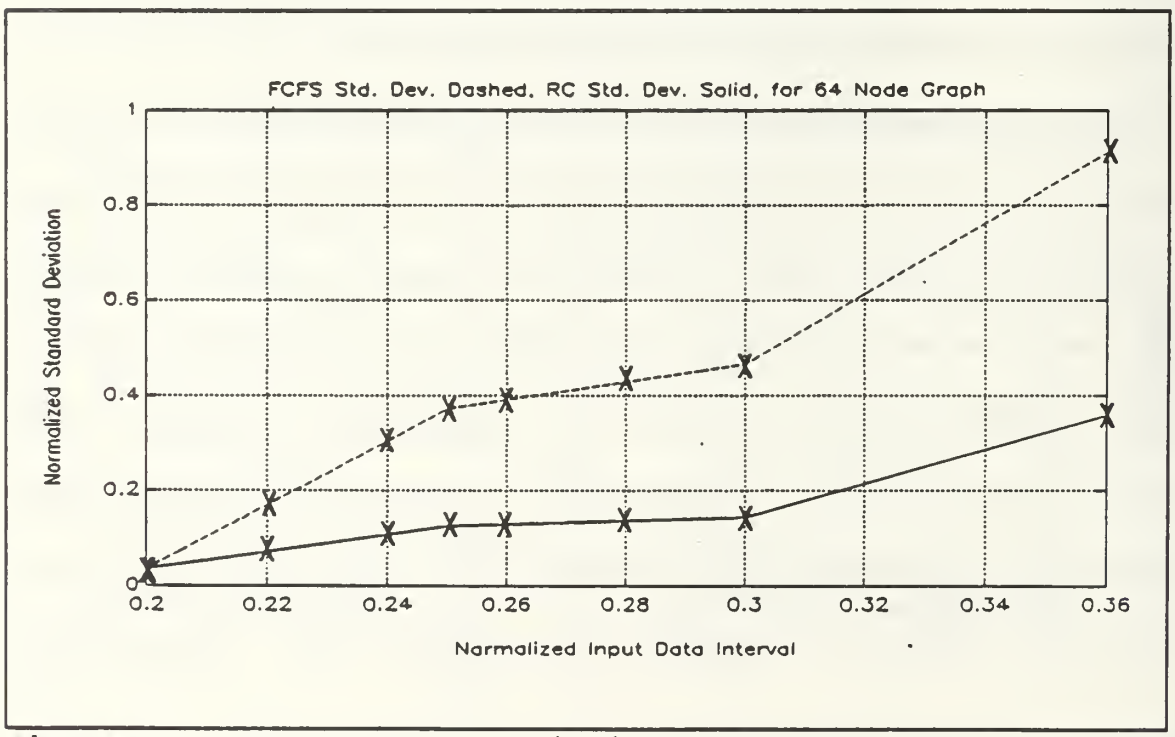

Figure 33: FFT Standard Deviation Blow Up

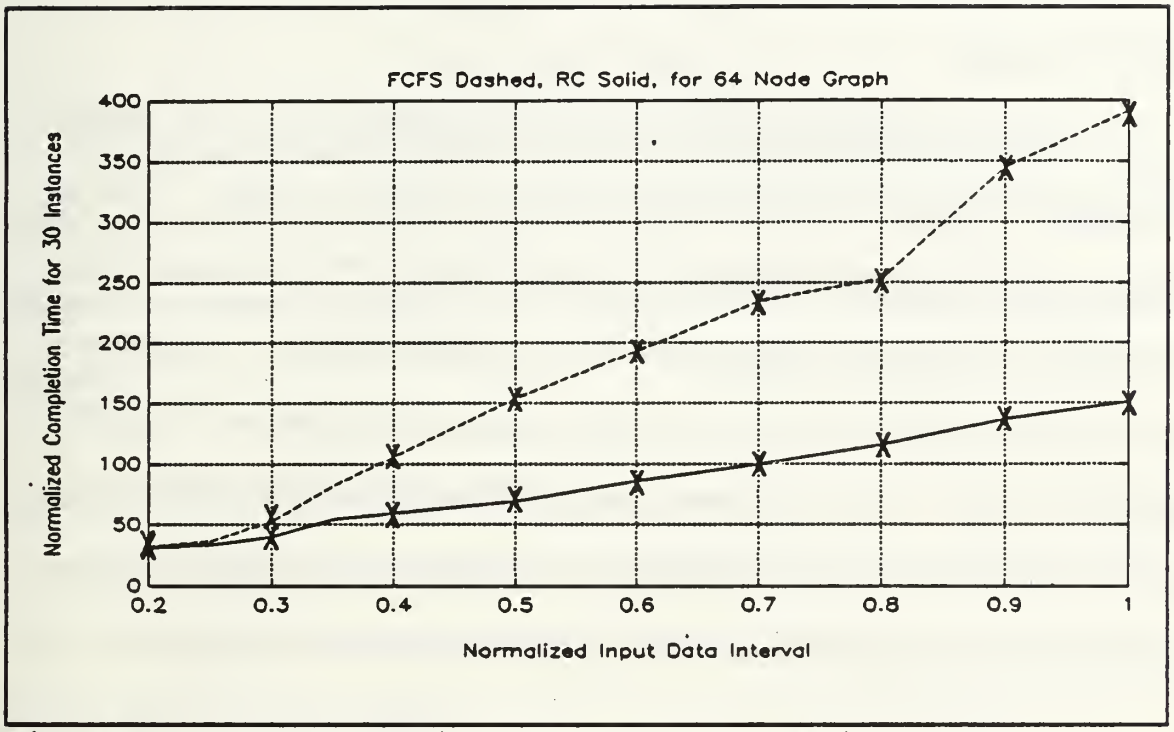

Figure 34: FFT Normalized Instance Completion Output

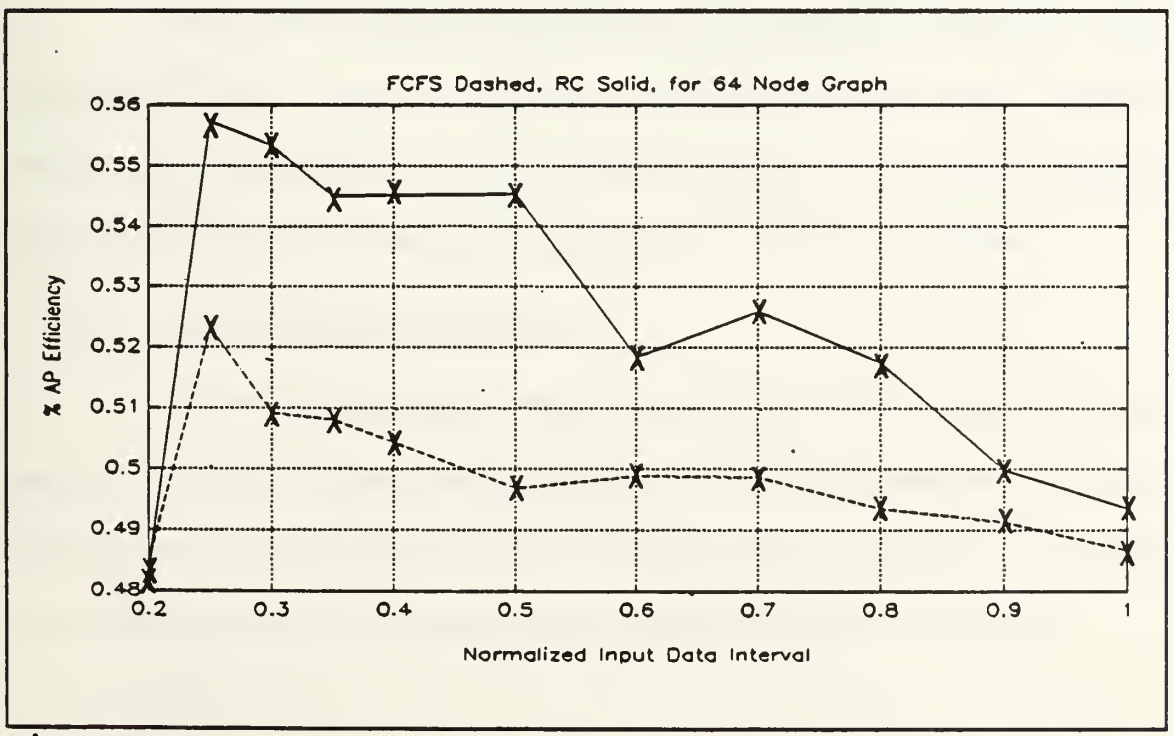

Figure 35: FFT AP Efficiency

graph. The low values are caused by the communication overhead involved in processing this type of graph.

The RC approach possesses a greater AP efficiency due to the assigned dependencies limiting the communication traffic on the DTN and CBUS. The RC approach normalized mean reaches unity five percent before the FCFS normalized mean. Additionally, the normalized standard deviation is consistently less than 0.5 regardless of load level. This implies that a much more uniform output results from the RC algorithm regardless of load. Since the observed normalized completion time for thirty graph instances is also less for the RC approach than the FCFS approach, it can be concluded that the RC approach performs well with communication intensive applications.

VI. CONCLUSIONS

Due to the increased complexity of Navy sensor systems and the increased signal processing requirements expected within the next ten years, a different scheduling approach may extend the lifetime of the current AN/UYS-2 hardware without major expense.

A. SUCCESS OF THE RC APPROACH

The performance results obtained using a simulation of simple applications indicate the effectiveness of the proposed technique in improving performance and predictability for communication bound graphs.

1. Communication Intensive

Communication intensive applications such as the sixty-four node FFT graph analyzed in Chapter V document an extensive improvement by use of the RC algorithm over that of FCFS scheduling. The RC approach produces a more uniform throughput while improving the AP efficiency and limiting the communication overhead.

2. Non-Communication Intensive

The non-communication intensive applications, like the correlator graph analyzed in Chapter V, demonstrate improvement dependent on the node execution times. For nodes with uniform execution times, no substantial improvement was noted. But for non-uniform execution times, quicker, more uniform throughput was observed.

Since chaining of nodes is a matter of trial and error in FCFS scheduling, and unrestrained chaining results in loss of parallelism which can be detrimental to uniform AP utilization and throughput, the RC schedule offers a reasonable approach that would limit the work of the application programmer.

B. IMPROVING RC

If chaining is specified within the framework of RC assignment, its effect can be accurately predicted. An algorithm to use chaining in RC assignment is being developed. Since the cylinder creates a semblance of assigning nodes to AP's even though it is not enforced, use of this knowledge can absorb even the communication overheads involved between nodes.

Given a specific assignment, it is known which queues are accessed at the same time. This information can be used to algorithmically assign GM's to queues so that each GM operates at the maximum possible bandwidth.

The current RC algorithm can introduce several different control token sets. Establishment of criteria to select the minimal set of dependencies needs to be developed.

It is also required that the AN/UYS-2 support on-line reconfiguration of the PGM graph for an application. This reconfiguration is typically performed by the operator by removing or adding one or more branches of the PGM graphs. In RC assignment, new nodes could be assigned in the empty slots so as to leave the rest of the assignment undisturbed.

C. PROPOSED RESEARCH

1. Hardware Modifications

Hardware modifications provide the long term solution to ensuring that the AN/UYS-2 parallel signal processor remains in the Navy forefront. The modular design of the AN/UYS-2 makes hardware modification an attractive long term solution.

a. Systolic Array Processor

Systolic architectures began to be used during the 1980's [EVANS 82, pp. 343]. Each PE corresponds to one binary tree level in the systolic array [MOSCOVITZ 90, pp. 355-357, KUNG 85, pp. 128-131]. Inclusion of a systolic array processor will expand the functional capability and increase the performance of the AN/UYS-2 by improving its processing capabilities.

b. Open Architecture

An open architecture would ensure that current "industry technology" could be inserted into the AN/UYS-2 without the extensive cost of modification. An open architecture can be obtained by continuing to develop the AN/UYS-2's modularity while maintaining a standard software communication interface.

2. Software Modifications

Despite the attractiveness of hardware solutions, the inexpensive and easy to implement software modifications could yield the same improvement in a shorter period of time.

a. User Friendly Processing Graph Methodology

Steps taken toward a more user friendly PGM would greatly speed up the developer's process. Although the current PGM is usable, the application developer must spend many hours researching the primitives and establishing the basics of the graph that he wishes to use.

b. Throughput Enhancements

The throughput enhancements discussed in this thesis can easily be implemented in software to achieve an improvement in the AN/UYS-2.

(1) Node Chaining

It was shown that the chaining of nodes under the FCFS algorithm demonstrates the same level of improvement as that of the RC algorithm without chaining.

(2) Scheduling

The RC scheduling algorithm offers an attractive choice to achieve the same results as chaining without the extra effort required by the application programmers. Implementation of this algorithm would not only yield an improvement on communication intensive graphs, but also on other applications. An unlimited or selective loss of incoming data arriving at a higher input rate than the maximum achievable by the AN/UYS-2 would not result in unreasonable delay times.

(3) Fault Tolerance

Fault tolerance refers to the malfunction of a resource element which prevents it from performing its function. In the AN/UYS-2 this fault tolerance is handled by the CPP which *removes* the associate malfunctioning element from the machine. However, while this would still work in the existing approach, the RC technique would lose a segment of its cylinder. It appears that this loss of performance could be kept temporary by switching to another cylinder with a larger circumference *on-line*. However, this issue needs further investigation.

APPENDIX A: REVOLVING CYLINDER CODE

```
// Description  : The code listed below follows the algorithms outlined in Chapter III.
//              : This procedure deals with the initial assigning of nodes to the Cylinder.
//              : Nothing else need be said.
// Parameters   : tempgnodelisting   - the node graph itself
//              numaps               - the number of aps in the AN/UYS-2
dependencyqs *topgraph :: assignrc(gnode *tempgnodelisting,int numaps) {
   gnode         *temp2gnodelisting = tempgnodelisting;
   gnode         *temp3gnodelisting = NULL;
   ptrtoptrtoaq  *tempptrtoptr = NULL;
   topgraph           *q = NULL;
   topgraph           *qtemp = NULL;
   topgraph           *q2temp = NULL;
   topgraph           *temp = NULL;
   int           count = 0;
   long int      circumference = 0;
   long int      circum2ference = 0;
   long int      maxwidth = 0;
   long int      minwidth = 1000000;
   long int      widthavg = 0;
   long int           j = 0;
   long int      t = 0;
   cyltype       *cylinder = NULL;
   cyltype       *tempcylinder = NULL;
   cylentrytype  *cyl2entrylist = NULL;
   boolean       iopinnode = false;
   boolean       iopoutnode = false;
   dependencyqs      *headofdepqs = NULL;
   while (temp2gnodelisting != NULL) {
     tempptrtoptr = tempgnodelisting->getgnodeinputqslist(temp2gnodelisting->
                          getnodeid());
     while (tempptrtoptr != NULL) {
       if (tempptrtoptr->getnodein() == -1) {
          iopinnode = true;
          break;
       }
       else {
          iopinnode = false;
```

```
  };
  tempptrtoptr = tempptrtoptr->getnextelement();
};
tempptrtoptr = tempgnodelisting->getgnodeoutputqslist(temp2gnodelisting->
                    getnodeid());
while (tempptrtoptr != NULL) {
  if (tempptrtoptr->getnodeout() == -1) {
    iopoutnode = true;
    break;
  }
  else {
    iopoutnode = false;
  };
  tempptrtoptr = tempptrtoptr->getnextelement();
};
if ((iopinnode) || (iopoutnode)) { // do nothing
}
else {
  count++;
  if (q == NULL) {
    if (!(q = new topgraph)) {
      fprintf(stderr,"Insufficient memory for topgraph\n");
      exit(1);
    };
    q->id = temp2gnodelisting->getnodeid();
    q->width = tempgnodelisting->getprimtime(q->id);
    if (q->width < minwidth) {
      minwidth = q->width;
    };
    if (q->width > maxwidth) {
      maxwidth = q->width;
    }; // note q->est set equal to zero by constructor
    circumference = circumference + q->width;
    qtemp = q;
  }
  else {
    if (!(qtemp->next = new topgraph)) {
      fprintf(stderr,"Insufficient memory for topgraph\n");
      exit(1);
    };
    qtemp->next->id = temp2gnodelisting->getnodeid();
    qtemp->next->width = tempgnodelisting->getprimtime(qtemp->next->id);
    if (qtemp->next->width < minwidth) {
```

```
            minwidth = qtemp->next->width;
          };
          if (qtemp->next->width > maxwidth) {
            maxwidth = qtemp->next->width;
          };
          circumference = circumference + qtemp->next->width;
          qtemp = qtemp->next;
        };
      };
      temp2gnodelisting = temp2gnodelisting->getnextgnode();
};
widthavg = minwidth;
circumference = circumference / numaps;
circum2ference = circumference;
while (maxwidth > circum2ference) {
  circum2ference = circum2ference + widthavg;
};
circumference = circum2ference;
for (int i=1;i<=numaps;i++) {
  if (cylinder == NULL) {
    if (!(cylinder = new cyltype)) {
      fprintf(stderr,"Insufficient memory for cyltype\n");
      exit(1);
    };
    j = 0;
    while (j < circumference) {
      if (cylinder->cylentrylist == NULL) {
        if (!(cylinder->cylentrylist = new cylentrytype)) {
          fprintf(stderr,"Insufficient memory for cylentrytype\n");
          exit(1);
        };
        cylinder->cylentrylist->widthstarttime = j;
        cyl2entrylist = cylinder->cylentrylist;
      }
      else {
        if (!(cyl2entrylist->nextcylentry = new cylentrytype)) {
          fprintf(stderr,"Insufficient memory for cylentrytype\n");
          exit(1);
        };
        cyl2entrylist->nextcylentry->widthstarttime = j;
        cyl2entrylist = cyl2entrylist->nextcylentry;
      };
      j = j + widthavg;
```

```
    };
    tempcylinder = cylinder;
  }
  else {
    if (!(tempcylinder->nextcylap = new cyltype)) {
      fprintf(stderr,"Insufficient memory for cyltype\n");
      exit(1);
    };
    j = 0;
    while (j < circumference) {
      if (tempcylinder->nextcylap->cylentrylist == NULL) {
        if (!(tempcylinder->nextcylap->cylentrylist = new cylentrytype)){
         fprintf(stderr,"Insufficient memory for cylentrytype\n");
         exit(1);
        };
        tempcylinder->nextcylap->cylentrylist->widthstarttime = j;
        cyl2entrylist = tempcylinder->nextcylap->cylentrylist;
      }
      else {
        if (!(cyl2entrylist->nextcylentry = new cylentrytype)) {
         fprintf(stderr,"Insufficient memory for cylentrytype\n");
         exit(1);
        };
        cyl2entrylist->nextcylentry->widthstarttime = j;
        cyl2entrylist = cyl2entrylist->nextcylentry;
      };
      j = j + widthavg;
    };
    tempcylinder = tempcylinder->nextcylap;
  };
};
qtemp = q;
while (qtemp != NULL) {
  temp = qtemp;
  qtemp = qtemp->next;
  if (cylinder->checkfreecylspace(cylinder) < (temp->width/widthavg)) {
    circumference = circumference + circumference;
    cylinder = cylinder->increasecylsize(cylinder,circumference,widthavg);
    qtemp = q;
    temp = qtemp;
    qtemp = qtemp->next;
  };
  t = schedulenode(temp,temp->est,circumference,widthavg,numaps,cylinder);
```

```
      tempptrtoptr = tempgnodelisting->getgnodeoutputqslist(temp->id);
      recestupdt(tempptrtoptr,qtemp,t+temp->width);
    };
    headofdepqs = cylinder->createdeps(cylinder,circumference,widthavg);
    return headofdepqs;
};
// Description  : The following procedure deals with scheduling a node in a slot in the
cylinder.
// Parameters    : temp              - the graph node to schedule
//                 t                 - the time to attempt to schedule at
//                 circum            - the circumference of the cylinder
//                 widthavg          - the width of a slot in the cylinder
//                 numaps            - the number of aps in the AN/UYS-2
//                 cyl               - the cylinder itself
long int topgraph :: schedulenode(topgraph *temp,long int t,long int circum,
                        long int widthavg,int numaps,cyltype *cyl) {
    cyltype       *tempcylinder = cyl;
    cylentrytype  *tempcyl,*temp2cyl;
    boolean       scheduled = false;
    boolean       available = false;
    long int      insertime = 0;
    int           index = 0;
    int           blockcount = 0;
    long int      oldinsertime = 0;
    insertime = t;
    oldinsertime = insertime;
    while (scheduled == false) {
      insertime = insertime % circum;
      if (insertime < oldinsertime) {
        index--;
      };
      tempcylinder = cyl;
      for (int i=1;i<=numaps;i++) {
        tempcyl = tempcylinder->cylentrylist;
        while ((insertime > tempcyl->widthstarttime) && (tempcyl != NULL)) {
          tempcyl = tempcyl->nextcylentry;
        };
        if ((temp->width % widthavg) == 0) {
          blockcount = temp->width / widthavg;
        }
        else {
          blockcount = temp->width / widthavg + 1;
        };
```

```
      temp2cyl = tempcyl;
      for (int j=1;j<=blockcount;j++) {
        if (temp2cyl == NULL) {
          // at end of current circum so lets check back at beginning
          temp2cyl = tempcylinder->cylentrylist;
        };
        if (temp2cyl->nodesch == 0) {
          available = true;
        }
        else {
          available = false;
          break;
        };
        temp2cyl = temp2cyl->nextcylentry;
      };
      if (available) {
        temp2cyl = tempcyl;
        for (int j=1;j<=blockcount;j++) {
          if (temp2cyl == NULL) {
           // at end of current circum
           temp2cyl = tempcylinder->cylentrylist;
          };
          temp2cyl->nodesch = temp->id;
          temp2cyl->nodeicount = index;
          temp2cyl = temp2cyl->nextcylentry;
        };
        scheduled = true;
        break;
      }
      else {
        tempcylinder = tempcylinder->nextcylap;
      };
    };
    if (scheduled == false) {
      oldinsertime = insertime;
      insertime = (insertime + widthavg) % circum;
    };
  };
  return insertime;
};
// Description  : Checks the free space remaining in the cylinder by checking
//                to see if a node other than zero has been scheduled into that
//                cylinder slot.
```

```
// Parameters   : tempcylinder        - The cylinder
long int cyltype :: checkfreecylspace(cyltype *tempcylinder) {
   cylentrytype  *tempcylentry;
   long int      maxfree = 0;
   long int      currfree = 0;
   long int      startfree = 0;
   boolean       atstartofapcyl = true;
   while (tempcylinder != NULL) {
      tempcylentry = tempcylinder->cylentrylist;
      atstartofapcyl = true;
      startfree = 0;
      currfree = 0;
      while (tempcylentry != NULL) {
        if ((tempcylentry->nodesch == 0) && (atstartofapcyl)) {
           startfree++;
           if (startfree > maxfree) {
              maxfree = startfree;
           };
        }
        else {
           atstartofapcyl = false;
           if (tempcylentry->nodesch == 0) {
              currfree++;
              if (currfree > maxfree) {
               maxfree = currfree;
              };
           }
           else {
              currfree = 0;
           };
           if ((tempcylentry->nextcylentry == NULL) && (currfree > 0)) {
              if ((currfree + startfree) > maxfree) {
               maxfree = currfree + startfree;
              };
           };
        };
        tempcylentry = tempcylentry->nextcylentry;
      };
      tempcylinder = tempcylinder->nextcylap;
   };
   return maxfree;
};
// Description  : Increments the size of the cylinder by circum and resets its
```

```
//              slots to reflect no nodes scheduled.
// Parameters   : tempcylinder        - The cylinder itself
//              circum              - The size to increase the cylinder to
//              minwidthavg         - The cylinder slot size
cyltype *cyltype :: increasecylsize(cyltype *tempcylinder,long int circum,
                        long int minwidthavg) {
  cyltype       *temp2cylinder = tempcylinder;
  cylentrytype  *tempcylentry;
  long int      blockest = 0;
  while (temp2cylinder != NULL) {
    blockest = 0;
    tempcylentry = temp2cylinder->cylentrylist;
    while (tempcylentry->nextcylentry != NULL) {
      tempcylentry->nodesch = 0;
      blockest = blockest + minwidthavg;
      tempcylentry = tempcylentry->nextcylentry;
    };
    blockest = blockest + minwidthavg;
    tempcylentry->nodesch = 0;
    while (blockest < circum) {
      tempcylentry->nextcylentry = new cylentrytype;
      tempcylentry->nextcylentry->widthstarttime = blockest;
      tempcylentry = tempcylentry->nextcylentry;
      blockest = blockest + minwidthavg;
    };
    temp2cylinder = temp2cylinder->nextcylap;
  };
  return tempcylinder;
};
// Description  : Returns true if the searched node is an ancestor of the
//              queue pointed to by temp2ptrtoptr output node.
// Parameters   : temp2ptrtoptr       - ptrtoptr to a queue to check nodes
boolean cyltype :: ancestor(ptrtoptrtoaq *temp2ptrtoptr,int citprime) {
  boolean       questanc = false;
  ptrtoptrtoaq  *temp3ptrtoptr;
  while (temp2ptrtoptr != NULL) {
    if (questanc == true) {
      break;
    };
    if (temp2ptrtoptr->getnodeout() == -1) {
      // at end of chain so do nothing, this should work here since
      // here we are dealing with the graph itself and not the cylinder
    }
```

```
      else {
        if (citprime == temp2ptrtoptr->getnodeout()) {
          questanc = true;
          break;
        };
        temp3ptrtoptr = gnodelisting->getgnodeoutputqslist(temp2ptrtoptr->getnodeout());
        questanc = ancestor(temp3ptrtoptr,citprime);
      };
      temp2ptrtoptr = temp2ptrtoptr->getnextelement();
    };
    return questanc;
};
// Description  : The following procedure creates the dependencies among nodes in the
cylinder.
// Paramaters    : cylinder- the cylinder
//               : circum - the circumference of the cylinder
//               : widthavg    - the width of a slot in the cylinder
dependencyqs *cyltype :: createdeps(cyltype *cylinder,long int circum,
                        long int widthavg) {
    ptrtoptrtoaq  *temp2ptrtoptr,
                  *temp3ptrtoptr;
    cyltype       *tempcylinder;
    cylentrytype  *tempcylentry,
                  *tempprimecylentry;
    dependencylist    *tempptrtodepptr;
    dependencyqs      *headdepq = NULL;
    dependencyqs      *tempheaddepq;
    long int      t,
                  tprime,
                  maxentrytotry,
                  entrycount;
    boolean       cicircumvalid = false;
    boolean       already dep = false;
    boolean       notable = true;
    gnode         *tempgnodelisting;
    tempcylinder = cylinder;
    while (tempcylinder != NULL) {
      tempcylentry = tempcylinder->cylentrylist;
      tempprimecylentry = tempcylinder->cylentrylist;
      t = 0;
      while (t < circum) {
        tempprimecylentry = tempcylinder->cylentrylist;
        tprime = t + widthavg;
```

```
while (tprime < circum) {
  while (tempcylentry->widthstarttime != t) {
    tempcylentry = tempcylentry->nextcylentry;
  };
  while (tempprimecylentry->widthstarttime != tprime) {
    tempprimecylentry = tempprimecylentry->nextcylentry;
  };
  if ((tempcylentry->nodesch >= tempprimecylentry->nodesch) ||
        (tempcylentry->nodesch == 0)) {
    // do nothing same node in next block
    // of cylinder or not currently a node scheduled in this block
  }
  else {
    tempheaddepq = headdepq;
    while ((tempheaddepq != NULL) && (tempheaddepq->nodefrom !=
        tempcylentry->nodesch) && (tempheaddepq->nodeto !=
     tempprimecylentry->nodesch)) {
     tempheaddepq = tempheaddepq->nextdepq;
    };
    if (tempheaddepq == NULL) { // not already a dependency for these
     temp2ptrtoptr = gnodelisting->getgnodeoutputqslist(tempcylentry->
                  nodesch);
     if ((tempcylentry->nodeicount == tempprimecylentry->nodeicount) &&
        (ancestor(temp2ptrtoptr,tempprimecylentry->nodesch)) == false) {
        if (tempcylentry->fromdepqs == NULL) {
         if (!(tempcylentry->fromdepqs = new dependencylist)) {
            fprintf(stderr,"Insufficient memory for dependlst\n");
            exit(1);
         };
         if (!(tempcylentry->fromdepqs->ptrtodepq = new dependencyqs)) {
            fprintf(stderr,"Insufficient memory for dependqs\n");
            exit(1);
         };
         tempcylentry->fromdepqs->ptrtodepq->nodefrom =
                  tempcylentry->nodesch;
         tempcylentry->fromdepqs->ptrtodepq->nodeto =
                  tempprimecylentry->nodesch;

         if (headdepq == NULL) {
            headdepq = tempcylentry->fromdepqs->ptrtodepq;
         }
         else {
            tempheaddepq = headdepq;
```

```
    while (tempheaddepq->nextdepq != NULL) {
    tempheaddepq = tempheaddepq->nextdepq;
    };
    tempheaddepq->nextdepq = tempcylentry->fromdepqs->
            ptrtodepq;
 };

}
else {
 tempptrtodepptr = tempcylentry->fromdepqs;
 alreadydep = false;
 while ((tempptrtodepptr->ptrtonextptrtodepq != NULL) &&
      (alreadydep == false)) {
   if (tempptrtodepptr->ptrtodepq->nodeto ==
          tempprimecylentry->nodesch) {
   // a dependency already exists for this pair so do NOT
   // create another one.
     alreadydep = true;
   }
   else {
     tempptrtodepptr = tempptrtodepptr->ptrtonextptrtodepq;
   };
 };
 if (alreadydep) {
   // do nothing since a dependency already exists for this
 }
 else {
   if (!(tempptrtodepptr->ptrtonextptrtodepq =new dependencylist)) {
     fprintf(stderr,"Insufficient memory for deplst\n");
     exit(1);
   };
   if (!(tempptrtodepptr->ptrtonextptrtodepq->ptrtodepq =
          new dependencyqs)) {
     fprintf(stderr,"Insufficient memory for depqs\n");
     exit(1);
   };
   tempptrtodepptr->ptrtonextptrtodepq->ptrtodepq->nodefrom
          = tempcylentry->nodesch;
   tempptrtodepptr->ptrtonextptrtodepq->ptrtodepq->nodeto =
          tempprimecylentry->nodesch;

   if (headdepq == NULL) {
     headdepq = tempptrtodepptr->ptrtodepq;
```

```
                }
                else {
                   tempheaddepq = headdepq;
                   while (tempheaddepq->nextdepq != NULL) {
                    tempheaddepq = tempheaddepq->nextdepq;
                   };
                   tempheaddepq->nextdepq = tempptrtodepptr->ptrtodepq;
                };
             };
            };
            if (alreadydep == false) {
             if (tempprimecylentry->todepqs == NULL) {
                if (!(tempprimecylentry->todepqs = new dependencylist)) {
                   fprintf(stderr,"Insufficient memory for deplst\n");
                   exit(1);
                };
                tempprimecylentry->todepqs->ptrtodepq = tempheaddepq->
                      nextdepq;
             }
             else {
                tempptrtodepptr = tempprimecylentry->todepqs;
                while (tempptrtodepptr->ptrtonextptrtodepq != NULL) {
                   tempptrtodepptr = tempptrtodepptr->ptrtonextptrtodepq;
                };
                if (!(tempptrtodepptr->ptrtonextptrtodepq =
                         new dependencylist)) {
                   fprintf(stderr,"Insufficient memory for deplst\n");
                   exit(1);
                };
                tempptrtodepptr->ptrtonextptrtodepq->ptrtodepq =
                         tempheaddepq->nextdepq;
             };
            };
          };
         };
       };
       tprime = tprime + widthavg;
     };
     t = t + widthavg;
   };
   tempcylinder = tempcylinder->nextcylap;
};
// add a dependency from end to all input nodes
```

```
  tempcylinder = cylinder;
  maxentrytotry = circum / widthavg;
  while (cicircumvalid == false) {
    tempcylentry = tempcylinder->cylentrylist;
    entrycount = 0;
    while ((tempcylentry->nextcylentry != NULL) && (entrycount < maxentrytotry))
{
      entrycount++;
      tempcylentry = tempcylentry->nextcylentry;
    };
    if ((tempcylentry->nodesch != 0) && (tempcylentry->nodeicount == 0)) {
      cicircumvalid = true;
    }
    else {
      tempcylinder = tempcylinder->nextcylap;
      if (tempcylinder == NULL) {
        maxentrytotry = maxentrytotry - 1;
        tempcylinder = cylinder;
      };
    };
  };
  tempgnodelisting = gnodelisting;
  while (tempgnodelisting != NULL) {
    temp2ptrtoptr = tempgnodelisting->getgnodeinputqslist(tempgnodelisting->
                    getnodeid());
    while (temp2ptrtoptr != NULL) {
      if (temp2ptrtoptr->getnodein() == -1) {
      //iop node so get the next node following the iopnode
       temp3ptrtoptr = tempgnodelisting->getgnodeoutputqslist(tempgnodelisting->
                  getnodeid());
       while (temp3ptrtoptr != NULL) {
         // set dependencies to all these nodes from the end of the cylinder
         if (tempcylentry->fromdepqs == NULL) {
          if (!(tempcylentry->fromdepqs = new dependencylist)) {
            fprintf(stderr,"Insufficient memory for dependlist\n");
            exit(1);
          };
          if (!(tempcylentry->fromdepqs->ptrtodepq = new dependencyqs)) {
            fprintf(stderr,"Insufficient memory for dependencyqs\n");
            exit(1);
          };
          tempcylentry->fromdepqs->ptrtodepq->nodefrom = tempcylentry->
                    nodesch;
```

```
tempcylentry->fromdepqs->ptrtodepq->nodeto = temp3ptrtoptr->
                getnodeout();
tempcylentry->fromdepqs->ptrtodepq->deptokensize = 1;
if (headdepq == NULL) {
  headdepq = tempcylentry->fromdepqs->ptrtodepq;
}
else {
  tempheaddepq = headdepq;
  while (tempheaddepq->nextdepq != NULL) {
    tempheaddepq = tempheaddepq->nextdepq;
  };
    tempheaddepq->nextdepq = tempcylentry->fromdepqs->ptrtodepq;
};
}
else {
tempptrtodepptr = tempcylentry->fromdepqs;
already dep = false;
while ((tempptrtodepptr->ptrtonextptrtodepq != NULL) &&
     (already dep == false)) {
  if (tempptrtodepptr->ptrtodepq->nodeto == temp3ptrtoptr->
                getnodeout()){
    // a dependency already exists for this pair so do NOT
    // create another one.
    already dep = true;
  }
  else {
    tempptrtodepptr = tempptrtodepptr->ptrtonextptrtodepq;
  };
};
if (already dep) {
  // do nothing since a dependency already exists for this
}
else {
  if (!(tempptrtodepptr->ptrtonextptrtodepq = new dependencylist)) {
    fprintf(stderr,"Insufficient memory for deplist\n");
    exit(1);
  };
  if (!(tempptrtodepptr->ptrtonextptrtodepq->ptrtodepq = new
                dependencyqs)){
    fprintf(stderr,"Insufficient memory for depqs\n");
    exit(1);
  };
  tempptrtodepptr->ptrtonextptrtodepq->ptrtodepq->nodefrom =
```

```
                                tempcylentry->nodesch;
                tempptrtodepptr->ptrtonextptrtodepq->ptrtodepq->nodeto =
                                temp3ptrtoptr->getnodeout();
                tempptrtodepptr->ptrtonextptrtodepq->ptrtodepq->deptokensize = 1;
                if (headdepq == NULL) {
                  headdepq = tempptrtodepptr->ptrtonextptrtodepq->ptrtodepq;
                }
                else {
                  tempheaddepq = headdepq;
                  while (tempheaddepq->nextdepq != NULL) {
                   tempheaddepq = tempheaddepq->nextdepq;
                  };
                  tempheaddepq->nextdepq = tempptrtodepptr->
                                ptrtonextptrtodepq->ptrtodepq;
                };
              };
            };
            temp3ptrtoptr = temp3ptrtoptr->getnextelement();
          };
        };
        temp2ptrtoptr = temp2ptrtoptr->getnextelement();
      };
      tempgnodelisting = tempgnodelisting->getnextgnode();
    };
    tempheaddepq = headdepq;
    printf("\nThe following dependencies have been assigned:\n");
    while (tempheaddepq != NULL) {
      printf("From: ");
      printf("%d",tempheaddepq->nodefrom);
      printf(" To: ");
      printf("%d",tempheaddepq->nodeto);
      printf("\n");
      tempheaddepq = tempheaddepq->nextdepq;
    };
    return headdepq;
};
// Description  : Increments or decrements the tokens as required based on the
//                call, depending on whether or not it is at the head or tail
//                of the dependency.
// Parameters   : tempdeplist        - the list of dependencies
//                idnode             - the id number of the node to adjust
void dependencyqs :: adjusttokens(dependencyqs *tempdeplist,int idnode) {
  while (tempdeplist != NULL) {
```

```
    if (tempdeplist->nodefrom == idnode) {
      (tempdeplist->deptokensize)++;
    }
    else {
      if (tempdeplist->nodeto == idnode) {
        (tempdeplist->deptokensize)--;
      };
    };
    tempdeplist = tempdeplist->nextdepq;
  };
};
// Description  : Checks to see if the token condition of greater than zero is
//                satisfied.
// Parameters   : tempdeplist        - the list of dependencies
//                idnode             - the id number of the node to adjust
boolean dependencyqs :: checktokens(dependencyqs *tempdeplist,int idnode) {
  boolean oktoexec = true;
  while (tempdeplist != NULL) {
    if (tempdeplist->nodeto == idnode) {
      if (tempdeplist->deptokensize > 0) {
        oktoexec = true;
      }
      else {
        oktoexec = false;
        break;
      };
    };
    tempdeplist = tempdeplist->nextdepq;
  };
  return oktoexec;
};
// Description  : For each daughter of the last node inserted into the cylinder
//                update the earliest start time.  Recursively calls daughters
//                of daughters until the end of the graph is reached.
// Parameters   : temp2ptrtoptr      - ptr to ptr to a queue to adjust
//                q2temp             - the graph with the rest of the nodes
//                tpluswidth         - the time to adjust est to
void topgraph :: recestupdt(ptrtoptrtoaq *temp2ptrtoptr,topgraph *q2temp,
                            long int tpluswidth) {
  ptrtoptrtoaq  *temp3ptrtoptr;
  topgraph          *q3temp;
  while (temp2ptrtoptr != NULL) {
    if (temp2ptrtoptr->getnodeout() == -1) {
```

```
      cerr << "\nERROR DURING RECESTUPDT, OUTPUTNODE IN
CYLINDER\n";
    }
    else {
      q3temp = q2temp;
      while ((q3temp->id != temp2ptrtoptr->getnodeout()) &&
           (q3temp->next != NULL) && (q3temp != NULL)){
        q3temp = q3temp->next;
      };
      if (q3temp->id == temp2ptrtoptr->getnodeout()) {
        if (tpluswidth > q3temp->est) {
          q3temp->est = tpluswidth;
        };
        temp3ptrtoptr = gnodelisting->getgnodeoutputqslist(q3temp->id);
        recestupdt(temp3ptrtoptr,q2temp,tpluswidth);
      }; //else q3temp->next == NULL so at the end of this chain
    };
    temp2ptrtoptr = temp2ptrtoptr->getnextelement();
  };
};
```

APPENDIX B: PGM REPRESENTATION CODE

```
// Description  : The following procedure documents the order of loading the queue data
//              : into the cylinder and is included here for that reason.
void gqueue :: loadqueues() {
   int       numqueues = 0;
   gqueue  *tempgqueuelisting = NULL;
   cin > > numqueues;
   for (int queueloop=1;queueloop< =numqueues;queueloop++) {
      if (gqueuelisting == NULL) {
        if (!(gqueuelisting = new gqueue)) {
           fprintf(stderr,"Insufficient memory for gqueue\n");
           exit(1);
        };
        cin > > gqueuelisting->gqueueid;
        // NOTE GMID IS ASSIGNED BY LOADGRAPH ON ADJUSTMENT OF PTRS
        cin > > gqueuelisting->nodein;
        cin > > gqueuelisting->nodeout;
        cin > > gqueuelisting->datarate;
        cin > > gqueuelisting->overthreshold;
        cin > > gqueuelisting->productionqty;
        cin > > gqueuelisting->overcapacity;
        tempgqueuelisting = gqueuelisting;
      }
      else {
        if (!(tempgqueuelisting->nextelement = new gqueue)) {
           fprintf(stderr,"Insufficient memory for gqueue\n");
           exit(1);
        };
        cin > > tempgqueuelisting->nextelement->gqueueid;
        cin > > tempgqueuelisting->nextelement->nodein;
        cin > > tempgqueuelisting->nextelement->nodeout;
        cin > > tempgqueuelisting->nextelement->datarate;
        cin > > tempgqueuelisting->nextelement->overthreshold;
        cin > > tempgqueuelisting->nextelement->productionqty;
        cin > > tempgqueuelisting->nextelement->overcapacity;
        tempgqueuelisting = tempgqueuelisting->nextelement;
      };
   };
};
```

```
// Description  : The following procedure documents the loading of the graph into the
simulator
//             : and is included here for that reason.
long int gnode :: loadgraph(int numiops,int numgms,int numaps,
                    ioprocessors *tempioplist) {
  int       numnodes = 0;
  gnode     *tempgnodelisting = NULL;
  int       numgmtoassign = 0;
  int       isiopnode = numaps;
  int       numiopnodes = 0;
  long int sumexectimes = 0;
  cin > > numnodes;
  for (int nodeloop=1;nodeloop< =numnodes;nodeloop++) {
    if (gnodelisting == NULL) {
      if (!(gnodelisting = new gnode)) {
        fprintf(stderr,"Insufficient memory for gnode\n");
        exit(1);
      };
      cin > > gnodelisting->nodeid;
      cin > > isiopnode;
      if (isiopnode != 0) {
        gnodelisting->iopidassigned = ((nodeloop+1) % numiops)+1;
        tempioplist->assignnodetoiop(tempioplist,gnodelisting->nodeid,
                          gnodelisting->iopidassigned);
        numiopnodes++;
      };
      cin > > gnodelisting->aissize;
      cin > > gnodelisting->primtime;
      sumexectimes = sumexectimes + gnodelisting->primtime;
      // Now update queue pointers
      numgmtoassign = ((nodeloop+1) % numgms)+1;
      gnodelisting->ptrtoinqlist = gnodelisting->ptrtoinqlist->establishinptrs(
                          gnodelisting->ptrtoinqlist,numgmtoassign);
      gnodelisting->ptrtooutqlist = gnodelisting->ptrtooutqlist->establishoutptrs(
                          gnodelisting->ptrtooutqlist);
      tempgnodelisting = gnodelisting;
    }
    else {
      if (!(tempgnodelisting->nextgnode = new gnode)) {
        fprintf(stderr,"Insufficient memory for gnode\n");
        exit(1);
      };
      cin > > tempgnodelisting->nextgnode->nodeid;
```

```
        cin >> isiopnode;
        if (isiopnode != 0) {
          tempgnodelisting->nextgnode->iopidassigned = ((nodeloop+1) %
numiops)+1;

tempioplist->assignnodetoiop(tempioplist,tempgnodelisting->nextgnode->nodeid,
                             tempgnodelisting->nextgnode->iopidassigned);
          numiopnodes++;
        };
        cin >> tempgnodelisting->nextgnode->aissize;
        cin >> tempgnodelisting->nextgnode->primtime;
        sumexectimes = sumexectimes + tempgnodelisting->nextgnode->primtime;
        // Now load queue data for queues associated with this node
        numgmtoassign = ((nodeloop+1) % numgms)+1;
        tempgnodelisting->nextgnode->ptrtoinqlist = tempgnodelisting->nextgnode->
                             ptrtoinqlist->establishinptrs(
                             tempgnodelisting->nextgnode->ptrtoinqlist->
                             numgmtoassign);
        tempgnodelisting->nextgnode->ptrtooutqlist = tempgnodelisting->nextgnode
                                ->ptrtooutqlist->establishoutptrs(
                                tempgnodelisting->nextgnode->ptrtooutqlist);
        tempgnodelisting = tempgnodelisting->nextgnode;
      };
    };
    return (sumexectimes);
};
```

APPENDIX C: MAIN SIMULATOR CODE

```
// The following constitutes the main program of the simulator
// Processiop is called first to process the input/output processors.
// Processbus is then called for cbus and dtn to process them.
// Processgm is then called to process the global memories.
// Processsch is then called to process the scheduler->
// Processap is then called to process aps.
main() {
  // The variables
  schprocessor *scheduler;
  ioprocessors *ioplist = NULL;
  gmprocessors *gmlist  = NULL;
  approcessors *aplist  = NULL;
  boolean      breakdownpriority = false;
  boolean      invalidrun = true;
  int          numberiops = 0;
  int          numbergms  = 0;
  int          numberaps  = 0;
  int          instancestart = 0;
  int          instancefinish = 0;
  int          runcase    = 0;
  long int     simtime    = 0;
  dependencyqs *headofdeplist = NULL;
  topgraph     *dummy = NULL;
  gnodelisting = NULL;
  gqueuelisting = NULL;
  cout << "This simulator expects the graph data to be in a file 'graph' and";
  cout << " the EMSP Structure to be in IOP,GM,AP Order in a file 'emspstru'\n";
  cout << "\nEnter the starting node instance for examination ";
  cin >> instancestart;
  cout << "\nEnter the completing node instance for examination ";
  cin >> instancefinish;
  cout << "\n(1) Enter 1 for FCFS scheduled run.";
  cout << "\n(2) Enter 2 for RC scheduled run.\n";
  cin >> runcase;
  if ((runcase < 1) || (runcase > 2)) {
    cout << "\nInvalid run entered assuming FCFS run\n";
    runcase = 1;
  };
```

```
// Establish the Simulator structure
if (freopen("emspstru","r",stdin) == NULL) {
   fprintf(stderr, "error redirecting stdin\n");
}
else {
   cin >> numberiops;
   ioplist = ioplist->establishiops(numberiops,ioplist);
   cin >> numbergms;
   gmlist = gmlist->establishgms(numbergms,gmlist);
   cin >> numberaps;
   aplist = aplist->establishaps(numberaps,aplist);
   fclose(stdin);
};
if (!(scheduler = new schprocessor)) {
   fprintf(stderr,"Insufficient memory for schprocessor\n");
   exit(1);
};
scheduler->setobjectid(1,numberaps);
// Establish the graph structure by reading in the data required
if (freopen("graph","r",stdin) == NULL) {
   fprintf(stderr,"error redirecting stdin\n");
}
else {
   gqueuelisting->loadqueues();
   simtime = gnodelisting->loadgraph(numberiops,numbergms,numberaps,ioplist);
   fclose(stdin);
};
simtime = simtime * instancefinish;
cout << "\nThe expected simulation time in order to ensure last node ";
cout << "\ninstance completion is: ";
cout << simtime;
cout << " microseconds.\n";
if (freopen("results","w",stdout) == NULL) {
   fprintf(stderr,"error redirecting stdout\n");
}
else {
   printf("The Simulation Time is: ");
   printf("%ld",simtime);
   printf("\n");
   if (runcase == 2) {
     headofdeplist = dummy->assignrc(gnodelisting,numberaps);
   };
   for (clock=0;clock<=simtime;clock++) {
```

```
        ioplist->processiop(ioplist);
        cbus.processbus(scheduler,ioplist,aplist,gmlist);
        for (int i=0;i<16;i++) {
           dtn[i].processbus(scheduler,ioplist,aplist,gmlist);
        };
        gmlist->processgm(gmlist,aplist);
        scheduler->processsch(instancestart,instancefinish,runcase,
                              headofdeplist);
        aplist->processap(aplist,breakdownpriority,scheduler,instancestart,
                          instancefinish);
      };
      aplist->calcaunotbusytime(aplist);
      gnodelisting->calcallnodeinstavgtime(instancestart,instancefinish);
      gqueuelisting->calcqueuetimes(instancestart,instancefinish);
      fclose(stdout);
   };
}
```

APPENDIX D: INPUT/OUTPUT PROCESSOR CODE

```
// Description  : For every iop do the following
//                generate instructions for any external inputs that require it
//                if there is a transfer in progress do nothing otherwise
//                check to see if currently processing and time to be done
//                processing, if it is then update processing status and place
//                information in queue.
//                if not processing, then get the next node from the head of the
//                queue and process it.
//                go to next iop
// Calls        :  getcurrinst          - to get the currentnodes instruction
//                 getgnodenum          - to get the currentnodes id number
//                 getgnodeinputqslist - to get the pointer to the nodes inputqs
//                 getgnodeoutputqslist- to get the pointer to the nodes outputqs
//                 setfields           - to set the new instructions fields
//                 getnextelement      - to get the next queue information
//                 updatebusytill      - to update the objects busytill time
//                 setinst             - to only change the instruction name
void ioprocessors :: processiop(ioprocessors *listofiops) {
  ioprocessors   *templistofiops = listofiops;
  list           *tempnodesassigned;

  while (templistofiops != NULL) {
     tempnodesassigned = templistofiops->nodesassigned;
     while (tempnodesassigned != NULL) {
       templistofiops->ioobject.generateinsts(tempnodesassigned);
       tempnodesassigned = tempnodesassigned->nextentry;
     };
     if (templistofiops->ioobject.xferinprogress()) {
       // do nothing
     }
     else {
       if ((templistofiops->ioobject.isprocessing()) &&
          (templistofiops->ioobject.finishtime())) {
          templistofiops->ioobject.setprocessing(false);
          templistofiops->ioobject.placeinqueue();
       };
       if (templistofiops->ioobject.isprocessing() == false) {
           if (templistofiops->ioobject.getnextnode()) {
```

```
            templistofiops->ioobject.processiopnode();
          };
        };
      };
      templistofiops = templistofiops->nextiop;
    };
};
// Description  : Determine the current instruction
//                Based on this current instruction update the fields
//                nad generate a new instruction if required.
// Calls        : getcurrinst          - to get the current instruction
//                getgnodenum          - to get the node number
//                getgnodeinputqslist  - to get the list of input queues
//                setfields            - to set the fields for the instruction
//                getgqueueid          - to get the queue id number
//                getgmid              - to get the gm id number
//                getnextelement       - to get the next element in the list
//                updatebusytill       - to update the busytill time
//                assignloc            - to assign the location
//                getqthresh           - to get the queue threshold value
//                getqofgnodenum       - to get the queue number
void object :: processiopnode() {
  ptrtoptrtoaq *gnqptr;
  int          gnodenum,
              numrequired = 0;
  objectnode   *tempnodeptr;
  tempnodeptr = currentnode;
  if (tempnodeptr != NULL) {
    switch (tempnodeptr->getcurrinst()) {
      case eis: {
        gnodenum = tempnodeptr->getgnodenum();
        gnqptr = gnodelisting->getgnodeinputqslist(gnodenum);
        while (gnqptr != NULL) {
          numrequired++;
          tempnodeptr->setfields(rq,gnodenum,gnqptr->getgqueueid(),gm,
                    gnqptr->getgmid(),iop,objectid);
          gnqptr = gnqptr->getnextelement();
          if (gnqptr != NULL) {
           if (tempnodeptr->nextnode == NULL) {
            if (!(tempnodeptr->nextnode = new objectnode)) {
             fprintf(stderr,"Insufficient memory for objectnode\n");
             exit(1);
            };
```

```
                tempnodeptr->nextnode->nextnode = NULL;
              };
              tempnodeptr = tempnodeptr->nextnode;
            };
          };
          setprocessing(true);
          updatebusytill(40+0.22*numrequired);
          break;
        };
        case aq: {
          tempnodeptr->setinst(cq);
          tempnodeptr->locto = tempnodeptr->locfrom;
          tempnodeptr->locfrom.assignloc(iop,objectid);
          setprocessing(true);
          updatebusytill(20+0.11*gqueuelisting->getqthresh(tempnodeptr->
                getqofgnodenum()));
          break;
        };
      };
    };
};
```

APPENDIX E: GLOBAL MEMORY CODE

```
// Description  : For every gm do the following
//                if there is a transfer in progress do nothing otherwise
//                check to see if currently processing and time to be done
//                processing, if it is then update processing status and place
//                information in queue.
//                if not processing, then get the next node from the head of the
//                queue and process it.
//                go to next gm
// Calls        : xferinprogress      - to determine if currently xfering data
//                isprocessing        - to determine if currently processing
//                finishtime          - to determine if completed processing
//                setprocessing       - to update the processing status
//                placeinqueue        - to place in cbus or dtn queue
//                getnextnode         - to get the next node from the head of q
//                processgmnode       - to process the node
void gmprocessors :: processgm(gmprocessors *listofgms,approcessors *listofaps) {
   gmprocessors *templistofgms = listofgms;
   while (templistofgms != NULL) {
      if (templistofgms->gmobject.xferinprogress()) {
        // do nothing
      }
      else {
        if ((templistofgms->gmobject.isprocessing()) &&
            (templistofgms->gmobject.finishtime())) {
           templistofgms->gmobject.setprocessing(false);
           templistofgms->gmobject.placeinqueue();
        };
        if (templistofgms->gmobject.isprocessing() == false) {
           if (templistofgms->gmobject.getnextnode()) {
              templistofgms->gmobject.processgmnode(listofaps);
           };
        };
      };
      templistofgms = templistofgms->nextgmproc;
   };
};
// Description  :  Get the currentnodes instruction
//                 Based on that nodes instruction generate another appropriate
```

```
//                instruction for it and update the fields involved.
// Calls        :  getcurrinst          - to get the currentnodes current inst
//                 setinst              - to set the instruction
//                 assignloc            - to assign the location
//                 getqthresh           - to get the queue threshold value
//                 getqofgnodenum       - to get the queue id number
//                 setgnodenum          - to set the new node number
//                 getqnodeoutnum       - to get the queue id number for out node
//                 addtolength          - to increase the size of that queue
//                 subfromlength        - to decrease the size of that queue
//                 updatebusytill       - to update the busytill time
void object :: processgmnode(approcessors *temp2aplist) {
  size tempsize = ut;
  gqueue *tempgqueuelist = gqueuelisting;
  switch (currentnode->getcurrinst()) {
    case sis: {
      currentnode->setinst(ais);
      currentnode->locfrom = currentnode->locto;
      currentnode->locto = currentnode->locassoc;
      setprocessing(true);
      updatebusytill(44);
      break;
    };
    case rq: {
      currentnode->locto = currentnode->locfrom;
      currentnode->setinst(aq);
      currentnode->locfrom.assignloc(gm,objectid);
      setprocessing(true);
      updatebusytill(10+0.11*tempgqueuelist->getqthresh(currentnode->
              getqofgnodenum()));
      break;
    };
    case wq: {
      if (currentnode->locfrom.getlocation() == ap) {
        temp2aplist->updatebdstatus(temp2aplist,currentnode->locfrom.
                        getlocationnum());
      };
      tempsize = tempgqueuelist->addtolength(currentnode->getqofgnodenum());
      if (tempsize == ot) {
        currentnode->locfrom = currentnode->locto;
        currentnode->setinst(qot);
        currentnode->locto.assignloc(sch,1);
        currentnode->setgnodenum(tempgqueuelist->getqnodeoutnum(currentnode->
```

```
                                    getqofgnodenum()));
            updatebusytill(12);
          }
          else if (tempsize == oc) {
            currentnode->locfrom = currentnode->locto;
            currentnode->setinst(qoc);
            currentnode->locto.assignloc(sch,1);
            currentnode->setgnodenum(tempgqueuelist->getqnodeoutnum(currentnode->
                                    getqofgnodenum()));
            updatebusytill(12);
          }
          else {
            currentnode->setinst(dest);
            updatebusytill(12);
          };
          setprocessing(true);
          break;
        };
        case cq: {
          if (currentnode->locfrom.getlocation() == ap) {
            temp2aplist->updatebdstatus(temp2aplist,currentnode->locfrom.
                                getlocationnum());
          };
          tempsize = tempgqueuelist->subfromlength(currentnode->
                                getqofgnodenum());
          if (tempsize == uc) {
            currentnode->setinst(quc);
            currentnode->locfrom = currentnode->locto;
            currentnode->locto.assignloc(sch,1);
            updatebusytill(12);
          }
          else {
            currentnode->setinst(dest);
            updatebusytill(10);
          };
          setprocessing(true);
          break;
        };
      };
    };
```

APPENDIX F: SCHEDULER CODE

```
// Description  : The following function follows the scheduler description in Chapter IV.
// Calls        : isprocessing          - to check the processing status
//                finishtime            - to see if time to complete processing
//                setprocessing         - to set the processing status
//                placeinqueue          - to place the instruction in the queue
//                getnextnode           - to get the next node to process
//                retcurrinst           - to get the current instruction
//                addtolist             - to add an item to a list
//                subfromlist           - to remove an item from the list
//                setcurrinst           - to set the current instruction
//                getgnodeinputqslist   - to get the list of input queues
//                getnodenum            - to get the node id number
//                questionot            - to determine if queue over threshold
//                getnextelement        - to get the next queue list element
//                getiopnumber          - to get the iop number
//                estsink               - to establish the sink node
//                generatematch         - to match ready node to free ap
void schprocessor :: processsch(int nodeinstart,int nodeinstfinish,
                        int runcase,dependencyqs *tempheaddeplist) {
   // No assignment or referencing a specific object required since there is
   // only one scheduler->
   list          *tempfreeaplist,
                 *tempreadynodelist,
                 *tempexeclist;
   ptrtoptrtoaq   *tempgnqptr;
   boolean        allot = true;
   boolean        match = false;
   boolean        alreadyexec = false;
   boolean        gensinknode = false;
   int            tempiopnum = 0;
   int            queuecount = 0;
   int            replications = 1;
   if (schobject.xferinprogress()) {
      // do nothing
   }
   else {
```

```
if ((schobject.isprocessing()) && (schobject.finishtime())) {
  schobject.setprocessing(false);
  schobject.placeinqueue();
};
if (schobject.isprocessing()==false) {
  if (schobject.getnextnode()==true) {
     switch (schobject.retcurrinst()) {
       case rfis: {
        freeaplist = freeaplist->addtolist(freeaplist,
                     schobject.getcurrfromlocnum());
        schobject.setcurrinst(dest);
        schobject.setprocessing(true);
        schobject.updatebusytill(17);
        break;
       };
       case quc: {
        if (inhibitedlist->inhibited(inhibitedlist,gqueuelisting->
            getqnodeinnum(schobject.getqueuenum()))) {
           inhibitedlist = inhibitedlist->subfromlist(inhibitedlist,
                       gqueuelisting->getqnodeinnum(schobject.
                       getqueuenum()));
           printf("NODE BACK UNDERCAPACITY: ");
           printf("%d",gqueuelisting->getqnodeinnum(schobject.getqueuenum()));
           printf(" at clock: ");
           printf("%ld",clock);
           printf("\n");
        };
        schobject.setcurrinst(dest);
        schobject.setprocessing(true);
        schobject.updatebusytill(10);
        break;
       };
       case qoc: {
        if (inhibitedlist->inhibited(inhibitedlist,gqueuelisting->
            getqnodeinnum(schobject.getqueuenum()))) {
            // do nothing here instruction set to dest later
        }
        else {
           inhibitedlist = inhibitedlist->addtolist(inhibitedlist,
                       gqueuelisting->getqnodeinnum(schobject.
                           getqueuenum()));
           printf("NODE OVERCAPACITY: ");
           printf("%d",gqueuelisting->getqnodeinnum(schobject.
```

```
            getqueuenum()));
        printf(" at clock: ");
        printf("%ld",clock);
        printf("\n");
      };
    };
    case qot: {
     if (inhibitedlist->inhibited(inhibitedlist,schobject.
                          getnodenum())) {
        // do nothing now checking to see if current node inhibited
     }
     else {
        tempgnqptr = gnodelisting->getgnodeinputqslist(schobject.
                    getnodenum());
        gnodelisting->incnumqsrec(schobject.getnodenum());
        while (tempgnqptr != NULL) {
         queuecount++;
         if (tempgnqptr->questionot()) {
            // do nothing allot already true
         }
         else {
            allot = false;
         };
         tempgnqptr = tempgnqptr->getnextelement();
        };
        if (gnodelisting->areallqsrec(schobject.getnodenum(),
                    queuecount)) {
         gnodelisting->setnumqsrec(schobject.getnodenum());
         tempiopnum = gnodelisting->getiopnumber(schobject.
                        getnodenum());
         if (tempiopnum == 0) {
            if (allot) {
               replications = gqueuelisting->getrepnumber(schobject.
                        getqueuenum());
               for (int i=1;i<=replications;i++) {
                readynodelist = readynodelist->addtolist(
                      readynodelist,schobject.
                      getnodenum());
                gnodelisting->calctimebtwnbdrl(schobject.
                      getnodenum(),
                      nodeinststart,
                      nodeinstfinish);
               gnodelisting->setinstance(schobject.getnodenum(),
```

```
                        nodeinststart,
                        nodeinstfinish,false);
                };
              };
            }
            else {
              // sink node
              if (allot) {
                gensinknode = true;
              };
            };
          };
        };
        schobject.setcurrinst(dest);
        schobject.setprocessing(true);
        schobject.updatebusytill(17);
        break;
      };
    };
    if (gensinknode) {
      schobject.estsink(tempiopnum,nodeinststart,nodeinstfinish);
    }
    else {
      tempfreeaplist = freeaplist;
      tempreadynodelist = readynodelist;
      while ((tempreadynodelist != NULL) && (match == false)){
       if (tempfreeaplist != NULL) {
          tempexeclist = executinglist;
          while (tempexeclist != NULL) {
           if (tempexeclist->getnumber() ==
              tempreadynodelist->getnumber()) {
              alreadyexec = true;
              break;
           }
           else {
              alreadyexec = false;
           };
           tempexeclist = tempexeclist->nextentry;
          };
          if (alreadyexec) {
           //do nothing since an instance of this node is already exec
          }
          else {
```

```
if (runcase == 2) {
   if (tempheaddeplist->checktokens(tempheaddeplist,
               tempreadynodelist->
               getnumber())) {
      tempheaddeplist->adjusttokens(tempheaddeplist,
               tempreadynodelist->
               getnumber());
      match = true;
      schobject.updatebusytill(3);
      schobject.generatematch(tempreadynodelist->
               getnumber(),tempfreeaplist
               ->getnumber());
      executinglist = executinglist->addtolist(
               executinglist,schobject.
               getnodenum());
      freeaplist = freeaplist->subfromlist(freeaplist,
               tempfreeaplist->
               getnumber());
      readynodelist = readynodelist->subfromlist(
            readynodelist,
            schobject.
            getnodenum());
      gnodelisting->settimearratap(schobject.
            getnodenum(),
            nodeinststart,
            nodeinstfinish);
      gnodelisting->calcnodeinsttime(schobject.
            getnodenum(),
            nodeinststart,
            nodeinstfinish);
      tempfreeaplist = tempfreeaplist->nextentry;
   }
   else {
     // do nothing since dependencies are not met
   };
}
else { // FCFS case
   match = true;
   schobject.updatebusytill(3);
   schobject.generatematch(tempreadynodelist->getnumber(),
            tempfreeaplist->getnumber());
   executinglist = executinglist->addtolist(executinglist,
            schobject.getnodenum());
```

```
            freeaplist = freeaplist->subfromlist(freeaplist,
                         tempfreeaplist->
                         getnumber());
            readynodelist = readynodelist->subfromlist(readynodelist,
                    schobject.getnodenum());
            gnodelisting->settimearratap(schobject.getnodenum(),
                         nodeinststart,
                         nodeinstfinish);
            gnodelisting->calcnodeinsttime(schobject.getnodenum(),
                         nodeinststart,
                         nodeinstfinish);
            tempfreeaplist = tempfreeaplist->nextentry;
          };
        };
      };
      tempreadynodelist = tempreadynodelist->nextentry;
    };
  };
}
else {
  tempfreeaplist = freeaplist;
  tempreadynodelist = readynodelist;
  while ((tempreadynodelist != NULL) && (match == false)){
    if (tempfreeaplist != NULL) {
     tempexeclist = executinglist;
     while (tempexeclist != NULL) {
       if (tempexeclist->getnumber() ==
        tempreadynodelist->getnumber()) {
        alreadyexec = true;
        break;
       }
       else {
        alreadyexec = false;
       };
       tempexeclist = tempexeclist->nextentry;
     };
     if (alreadyexec) {
       //do nothing since an instance of this node is already exec
     }
     else {
       if (runcase == 2) {
        if (tempheaddeplist->checktokens(tempheaddeplist,
                    tempreadynodelist->
```

```
                getnumber())) {
    tempheaddeplist->adjusttokens(tempheaddeplist,
                tempreadynodelist->
                getnumber());
    match = true;
    schobject.updatebusytill(10);
    schobject.setprocessing(true);
    schobject.generatematch(tempreadynodelist->
                getnumber(),tempfreeaplist
                ->getnumber());
    executinglist = executinglist->addtolist(
                executinglist,schobject.
                getnodenum());
    freeaplist = freeaplist->subfromlist(freeaplist,
                tempfreeaplist->
                getnumber());
    readynodelist = readynodelist->subfromlist(
            readynodelist,
            schobject.
            getnodenum());
    gnodelisting->settimearratap(schobject.
              getnodenum(),
              nodeinststart,
              nodeinstfinish);
    gnodelisting->calcnodeinsttime(schobject.
                getnodenum(),
                nodeinststart,
                nodeinstfinish);
    tempfreeaplist = tempfreeaplist->nextentry;
  }
  else {
    // do nothing since dependencies are not met
  };
 }
 else { // FCFS case
  match = true;
  schobject.generatematch(tempreadynodelist->getnumber(),
            tempfreeaplist->getnumber());
  schobject.setprocessing(true);
  schobject.updatebusytill(10);
  executinglist = executinglist->addtolist(executinglist,
            schobject.getnodenum());
  freeaplist = freeaplist->subfromlist(freeaplist,
```

```
                                tempfreeaplist->
                                getnumber());
                    readynodelist = readynodelist->subfromlist(readynodelist,
                                schobject.getnodenum());
                    gnodelisting->settimearratap(schobject.getnodenum(),
                                nodeinststart,
                                nodeinstfinish);
                    gnodelisting->calcnodeinsttime(schobject.getnodenum(),
                                nodeinststart,
                                nodeinstfinish);
                    tempfreeaplist = tempfreeaplist->nextentry;
                  };
                };
               };
               tempreadynodelist = tempreadynodelist->nextentry;
             };
           };
         };
       };
     };
```

APPENDIX G: ARITHMETIC PROCESSOR CODE

```
// Description  : For each ap perform the following
//                if the ap is currently transferring information do nothing
//                otherwise perform the following
//                check to see if the breakdown is complete and forward on info
//                check to see if the execution is complete and forward on to bd
//                check to see if setup is complete and forward on to execution
//                adjust breakdown and setup status
//                if control unit is not processing setup and breakdown then
//                get the next instruction and process it
//                go to next ap
// Called by    : Main
// Calls        : xferinprogress      - to determine if currently xfering data
//                isprocessing        - to determine if currently processing
//                finishtime          - to determine if completed processing
//                setprocessing       - to update the processing status
//                placeinqueue        - to place in cbus or dtn queue
//                placeinbreakdown    - to place in breakdown queue
//                destroynode         - to delete the current node
//                getexecnode         - to get the next node to execute
//                getnodenum          - to get the node number of this node
//                updatebusytill      - to update the busytill time
//                sendrfis            - to send the rfis to the scheduler
//                getnextinst         - to fetch the next instruction for cu
void approcessors :: processap(approcessors *listofaps,boolean bkdnpriority,
                      schprocessor *tempsched,int nodeinststart,
                      int nodeinstfinish) {
  approcessors *templistofaps = listofaps;
  int numnode;
  while (templistofaps != NULL) {
    if ((templistofaps->apsetup.xferinprogress()) ||
       (templistofaps->apbreakdown.xferinprogress())) {
       // do nothing
    }
    else {
        if ((templistofaps->numbdnodes == 0) &&
            (templistofaps->apbreakdownstatus)) {
            templistofaps->apbreakdownstatus = false; // NEXT LINE NEW
            tempsched->updateexeclist(templistofaps->apbdholdnodenum);
```

```
    gnodelisting->calctimeinstatap(templistofaps->apbdholdnodenum,
                        nodeinststart,nodeinstfinish);
};
if ((templistofaps->apbreakdown.isprocessing()) &&
    (templistofaps->apbreakdown.finishtime()) &&
    (templistofaps->apbreakdownstatus == false)) {
   templistofaps->apbreakdownstatus = true;
   templistofaps->apbdholdnodenum = templistofaps->apbreakdown.
                        getnodenum();
   templistofaps->apbreakdown.setprocessing(false);
   templistofaps->apbreakdown.placeinqueue();
};
if (templistofaps->apexecuting.isprocessing() == false) {
   (templistofaps->aunotbusytime)++;
};
if ((templistofaps->apexecuting.isprocessing()) &&
   (templistofaps->apexecuting.finishtime())) {
   if ((templistofaps->apbreakdownstatus) ||
     (templistofaps->apbreakdown.isprocessing())) {
     (templistofaps->aunotbusytime)++;
   };
   if ((templistofaps->apbreakdownstatus == false) &&
     (templistofaps->apbreakdown.isprocessing() == false)) {
     templistofaps->apexecuting.setprocessing(false);
     templistofaps->placeinbreakdown(templistofaps->apexecuting.
                        returnexecnode());//MODIFICATION
   };
};
if ((templistofaps->apsetup.isprocessing()) &&
    (templistofaps->apsetup.finishtime())) {
   if (templistofaps->apsetupstatus == notstarted) {
      templistofaps->apsetupstatus = inprogress;
      templistofaps->apsetup.setprocessing(false);
      templistofaps->apsetup.placeinqueue();
   }
   else {
      if (templistofaps->apsetupstatus == inprogress) {
       templistofaps->apsetup.setprocessing(false);
       templistofaps->apsetup.destroynode();
      }
      else  //setupstatus complete
       if (templistofaps->apexecuting.isprocessing() == false) {
          templistofaps->apsetupstatus = notstarted;
```

```
                templistofaps->apsetup.setprocessing(false);
                templistofaps->apexecuting.getexecnode(templistofaps->
                                apsetup.
                                getcurrnode());
                templistofaps->apexecuting.setprocessing(true);
                numnode = templistofaps->apexecuting.getnodenum();
                templistofaps->apexecuting.updatebusytill(gnodelisting->
                                  getprimtime(
                                  numnode));
                templistofaps->apexecuting.sendrfis();
              };
            };
        };
        if ((templistofaps->apsetup.isprocessing() == false) &&
            (templistofaps->apbreakdown.isprocessing() == false)) {
          templistofaps->getnextinst(bkdnpriority);
          if ((templistofaps->numsunodes == 0) &&
            (templistofaps->apsetupstatus == inprogress)) {
            templistofaps->apsetupstatus = complete;
          };
        };
    };
    templistofaps = templistofaps->nextapproc;
  };
};
// Description  :  Determine the current instruction
//                 if it is ais then do the following
//                   determine the node number
//                   get the input qs list that goes with that node number
//                   for every entry in that list do the following
//                     establish a new rq instruction
//                   update the busytill time
//                 if it is aq then do the following
//                   if it is the last one that we are waiting on then prepare
//                   it for destruction
//                   update the busytill time
//                 if it is dest then do the following
//                   determine the nodenumber
//                   get the list of output qs associated with that node
//                   for every entry in that list do the following
//                     generate a wq instruction
//                   get the list of input qs associated with that node
//                   for every entry in that list do the following
```

```
//                    generate a cq instruction
//                    update the busytill time
// Calls        :  getcurrinst          - to get the currentnodes instruction
//                 getgnodenum          - to get the currentnodes id number
//                 getgnodeinputqslist - to get the pointer to the nodes inputqs
//                 getgnodeoutputqslist- to get the pointer to the nodes outputqs
//                 getaissize           - to get the ais size in words
//                 setfields            - to set the new instructions fields
//                 getnextelement       - to get the next queue information
//                 updatebusytill       - to update the objects busytill time
//                 setinst              - to only change the instruction name
int object :: processapnode(int numrequired) {
  objectnode     *tempnodeptr;
  int            count,
                 holdnwis,
                 gnodenum;
  ptrtoptrtoaq   *gnqptr,
                 *tempgnqptr;

  tempnodeptr = currentnode;
  switch (tempnodeptr->getcurrinst()) {
    case ais:
    {
      gnodenum = tempnodeptr->getgnodenum();
      holdnwis = gnodelisting->getaissize(gnodenum);
      gnqptr = gnodelisting->getgnodeinputqslist(gnodenum);
      tempgnqptr = gnqptr;
      while (tempgnqptr != NULL) {
       numrequired++;
       tempnodeptr->setfields(rq,gnodenum,tempgnqptr->getgqueueid(),gm,
                              tempgnqptr->getgmid(),ap,objectid);
       tempgnqptr = tempgnqptr->getnextelement();
       if (tempgnqptr != NULL) {
          if (tempnodeptr->nextnode == NULL) {
             if (!(tempnodeptr->nextnode = new objectnode)) {
              fprintf(stderr,"Insufficient memory for objectnode\n");
              exit(1);
             };
             tempnodeptr->nextnode->nextnode = NULL;
          };
          tempnodeptr = tempnodeptr->nextnode;
       };
      };
```

```
  setprocessing(true);
  updatebusytill(40+0.22*holdnwis);
  break;
};
case aq:
{
  if (--numrequired == 0) {
    tempnodeptr->setinst(dest);
  };
  setprocessing(true);
  updatebusytill(20+0.11*gqueuelisting->getqthresh(tempnodeptr->
            getqofgnodenum()));
  break;
};
case dest:  //only here if instruction done executing coming from breakdn
{
  count=0;
  gnodenum = tempnodeptr->getgnodenum();
  gnqptr = gnodelisting->getgnodeoutputqslist(gnodenum);
  tempgnqptr = gnodelisting->getgnodeinputqslist(gnodenum);
  while (gnqptr != NULL) {
   count++;
   tempnodeptr->setfields(wq,gnodenum,gnqptr->getgqueueid(),gm,
                  gnqptr->getgmid(),ap,objectid);
   gnqptr = gnqptr->getnextelement();
   if (gnqptr != NULL) {
      if (tempnodeptr->nextnode == NULL) {
         if (!(tempnodeptr->nextnode = new objectnode)) {
          fprintf(stderr,"Insufficient memory for objectnode\n");
          exit(1);
         };
         tempnodeptr->nextnode->nextnode = NULL;
      };
   };
   if (tempnodeptr->nextnode == NULL) {
      if (!(tempnodeptr->nextnode = new objectnode)) {
         fprintf(stderr,"Insufficient memory for objectnode\n");
         exit(1);
      };
   };
   tempnodeptr = tempnodeptr->nextnode;
  };
  while (tempgnqptr != NULL) {
```

```
        count++;
        tempnodeptr->setfields(cq,gnodenum,tempgnqptr->getgqueueid(),gm,
                               tempgnqptr->getgmid(),ap,objectid);
        tempgnqptr = tempgnqptr->getnextelement();
        if (tempgnqptr != NULL) {
          if (tempnodeptr->nextnode == NULL) {
            if (!(tempnodeptr->nextnode = new objectnode)) {
              fprintf(stderr,"Insufficient memory for objectnode\n");
              exit(1);
            };
          };
        };
        tempnodeptr = tempnodeptr->nextnode;
      };
      setprocessing(true);
      updatebusytill(12+0.11*count);
      numrequired = count;
      break;
    };
  };
  return numrequired;
};
```

APPENDIX H: INTER-COMMUNICATION CODE

```
// Description  : If waiting on a transfer to complete then check to see if
//                that transfer is complete, otherwise commence xfer.  If not
//                waiting on xfer to complete then check to see if currently
//                processing a xfer and if it is complete then complete xfer.
//                if not processing a transfer, then get the next instruction
//                and process.
// Calls        : checklocsxfering     - to check locations xfering
//                commencxfer          - to begin the transfer
//                completexfer         - to complete the transfer
//                isprocessing         - to check the processing status
//                finishtime           - to check if time complete
//                getnextinstandprocess- to get the next inst and process it
void xferproc :: processbus(schprocessor *scheduler,ioprocessors *ioplist,
                            approcessors *aplist,gmprocessors *gmlist) {
  if (waitingonxferatlocs) {
    if (xferobject.checklocsxfering(scheduler,ioplist,aplist,gmlist)) { // do nothing
    }
    else {
      waitingonxferatlocs = false;
      xferobject.commencexfer(scheduler,ioplist,aplist,gmlist);
    };
  }
  else {
    if ((xferobject.isprocessing()) && (xferobject.finishtime())) {
      xferobject.completexfer(scheduler,ioplist,aplist,gmlist);
    };
    if (xferobject.isprocessing() == false) {
      getnextinstandprocess(scheduler,ioplist,aplist,gmlist);
    };
  };
};
void xferproc :: getnextinstandprocess(schprocessor *scheduler,ioprocessors
*ioplist,approcessors
                          *aplist,gmprocessors *gmlist) {
  if (xferobject.getnextnode()) {
    if (xferobject.checklocsxfering(scheduler,ioplist,aplist,gmlist)) {
      waitingonxferatlocs = true;
    }
```

```
        else {xferobject.commencexfer(scheduler,ioplist,aplist,gmlist);
        };
}; };
```

APPENDIX I: RESULT GENERATION CODE

```
// Description  : The code included in this appendix represents the code added to the
simulator
//             : to keep track of desired output data.
void gnode :: calcnodeinsttime(int nodenumber,int nodeinststart,
                        int nodeinstfinish) {
   gnode *tempgnodeptr = gnodelisting;
   while ((tempgnodeptr->nodeid != nodenumber) && (tempgnodeptr != NULL)) {
      tempgnodeptr = tempgnodeptr->nextgnode;
   };
   if (tempgnodeptr == NULL) {
      cerr << "\nERROR CALCULATING NODE INSTANCE TIME\n";
   }
   else {
      printf("Off RL node number: ");
      printf("%d",nodenumber);
      printf(" instance: ");
      printf("%d",tempgnodeptr->lastinstoffrl);
      printf(" Clock: ");
      printf("%ld",clock);
      printf("\n");
      if ((tempgnodeptr->lastinstoffrl >= nodeinststart) &&
         (tempgnodeptr->lastinstoffrl <= nodeinstfinish)) {
        tempgnodeptr->timenodeinstonrl[tempgnodeptr->lastinstoffrl -
                              nodeinststart] = clock - tempgnodeptr->
                              timeinstontorl[tempgnodeptr->
                              lastinstoffrl - nodeinststart];
      };
      (tempgnodeptr->lastinstoffrl)++;
   };
};
void gnode :: calcallnodeinstavgtime(int nodeinststart,int nodeinstfinish) {
   gnode *tempgnodeptr = gnodelisting;
   while (tempgnodeptr != NULL) {
      printf("\nNode ID: ");
      printf("%d",tempgnodeptr->nodeid);
      if (tempgnodeptr->iopidassigned != 0) {
        printf(" is an IOP Node\n");
      }
```

```
      else {
        printf("\n");
        // NOTE ARRAY IS ASSUMED TO BE IN BOUNDS
        for (int count=nodeinststart;count<nodeinstfinish;count++) {
          tempgnodeptr->avgtimeonrl = tempgnodeptr->timenodeinstonrl[count-
                          nodeinststart] + tempgnodeptr->avgtimeonrl;
          tempgnodeptr->avgtimeatap = tempgnodeptr->timenodeinstatap[count-
                          nodeinststart] + tempgnodeptr->avgtimeatap;
          printf("Instance: ");
          printf("%d",count);
          printf(" Time on rl: ");
          printf("%ld",tempgnodeptr->timenodeinstonrl[count - nodeinststart]);
          printf(" Time between SIS and BD Completion: ");
          printf("%ld",tempgnodeptr->timenodeinstatap[count - nodeinststart]);
          printf("\n");
        };
        tempgnodeptr->avgtimeonrl = tempgnodeptr->avgtimeonrl / (nodeinstfinish
                        - nodeinststart);
        tempgnodeptr->avgtimeatap = tempgnodeptr->avgtimeatap / (nodeinstfinish
                        - nodeinststart);
        printf("Average time on rl: ");
        printf("%ld",tempgnodeptr->avgtimeonrl);
        printf(" Average time between SIS and BD Completion: ");
        printf("%ld",tempgnodeptr->avgtimeatap);
        printf("\n");
      };
      tempgnodeptr = tempgnodeptr->nextgnode;
    };
};
void gqueue :: calcqueuetimes(int nodeinststart,int nodeinstfinish) {
    gqueue *tempgqueueptr = gqueuelisting;

    while (tempgqueueptr != NULL) {
      printf("\nQueue ID: ");
      printf("%d",tempgqueueptr->gqueueid);
      if (tempgqueueptr->nodein == -1) {
        printf(" is an external input queue\n");
      }
      else {
        if (tempgqueueptr->nodeout == -1) {
          printf(" is an external output queue\n");
        }
        else {
```

```
            printf("\n");
            // NOTE ARRAY IS ASSUMED TO BE IN BOUNDS
            for (int count=nodeinststart;count<nodeinstfinish;count++) {
               printf("Instance: ");
               printf("%d",count);
               printf(" Time between BD Completion and Successor on RL: ");
               printf("%ld",tempgqueueptr->timebtwnbdrl[count - nodeinststart]);
               printf("\n");
            };
         };
      };
      tempgqueueptr = tempgqueueptr->nextelement;
   };
};
void gnode :: settimearratap(int nodenumber,int nodeinststart,
                        int nodeinstfinish) {
   gnode *tempgnodeptr = gnodelisting;

   while ((tempgnodeptr->nodeid != nodenumber) && (tempgnodeptr != NULL)) {
      tempgnodeptr = tempgnodeptr->nextgnode;
   };
   if (tempgnodeptr == NULL) {
      cerr << "\nERROR SETTING ARRIVAL TIME AT AP\n";
   }
   else {
      if ((tempgnodeptr->lastinstoffrl >= nodeinststart) &&
         (tempgnodeptr->lastinstoffrl <= nodeinstfinish)) {
        tempgnodeptr->timeinstarratap[tempgnodeptr->lastinstoffrl -
                              nodeinststart] = clock;
      };
   };
};
void approcessors :: calcaunotbusytime(approcessors *tempaplist) {
   while (tempaplist != NULL) {
      printf("\nAP Number: ");
      printf("%d",tempaplist->apsetup.getobjectid());
      printf(" Time AP AU NOT Busy: ");
      printf("%ld",tempaplist->aunotbusytime);
      printf("\n");
      tempaplist = tempaplist->nextapproc;
   };
};
void gqueue :: setcompbdtime(int loctoset) {
```

```
  timecompbd[loctoset] = clock;
};
void ptrtoptrtoaq :: setcompbdtime(int loctoset) {
  ptrtoaq->setcompbdtime(loctoset);
};
void gqueue :: calccompbdrltime(int loctocalc) {
  timebtwnbdrl[loctocalc] = clock - timecompbd[loctocalc];
};
void ptrtoptrtoaq :: calccompbdrltime(int loctocalc) {
  ptrtoaq->calccompbdrltime(loctocalc);
};
void gnode :: calctimebtwnbdrl(int nodenumber,int nodeinststart,
                    int nodeinstfinish){
  gnode *tempgnodeptr = gnodelisting;
  ptrtoptrtoaq *tempptrtoptr;
  while ((tempgnodeptr->nodeid != nodenumber) && (tempgnodeptr != NULL)) {
    tempgnodeptr = tempgnodeptr->nextgnode;
  };
  if (tempgnodeptr == NULL) {
    cerr << "\nERROR CALCULATING TIME BETWEEN BREAKDOWN AND
        READYLIST\n";
  }
  else {
    tempptrtoptr = tempgnodeptr->ptrtoinqlist;
    if ((((tempgnodeptr->lastinstoffrl) - 1) >= nodeinststart) &&
      (((tempgnodeptr->lastinstoffrl) - 1) < nodeinstfinish)) {
      while (tempptrtoptr != NULL) {                    //-1
        tempptrtoptr->calccompbdrltime((tempgnodeptr->lastinstoffrl) -
                    nodeinststart);
        tempptrtoptr = tempptrtoptr->getnextelement();
      };
    };
  };
};
void gnode :: calctimeinstatap(int nodenumber,int nodeinststart,
                    int nodeinstfinish) {
  gnode *tempgnodeptr = gnodelisting;
  ptrtoptrtoaq *tempptrtoptr;
  while ((tempgnodeptr->nodeid != nodenumber) && (tempgnodeptr != NULL)) {
    tempgnodeptr = tempgnodeptr->nextgnode;
  };
  if (tempgnodeptr == NULL) {
    cerr << "\nERROR CALCULATING NODE INSTANCE TIME\n";
```

```
    }
    else {
      if ((((tempgnodeptr->lastinstoffrl) - 1) >= nodeinststart) &&
        (((tempgnodeptr->lastinstoffrl) - 1) <= nodeinstfinish)) {
        // NEXT FOUR LINES ARE NEW EXPERIMENTAL AS OF 8/28/91
        tempptrtoptr = tempgnodeptr->ptrtooutqlist;  // ??INQLIST OR OUTQLIST
        while (tempptrtoptr != NULL) {
          tempptrtoptr->setcompbdtime((tempgnodeptr->lastinstoffrl) - 1 -
                                      nodeinststart);
          tempptrtoptr = tempptrtoptr->getnextelement();
        };
        tempgnodeptr->timenodeinstatap[(tempgnodeptr->lastinstoffrl - 1) -
                                      nodeinststart] = clock - tempgnodeptr->
                                      timeinstarratap[(tempgnodeptr->
                                      lastinstoffrl - 1) - nodeinststart];
      };
    };
};
```

LIST OF REFERENCES

[APPLIC 90]
AT&T Technologies, Report CDRL Q001, *Enhanced Modular Signal Processor (EMSP) Application Programmer User Manual*, AT&T Bell Laboratories, 1 August 1990.

[ARVIND 80]
ARVIND, J., "Decomposing a Program for Multiple Processor Systems", *Proceedings of the 1980 International Conference on Parallel Processing*, August 1980.

[BEAUCHAMP 79]
Beauchamp, K., and Yuen, C., *Digital Methods For Signal Analysis*, George Allen Lunwin, 1979.

[BELLANGER 84]
Bellanger, M., *Digital Processing of Signals, Theory and Practice*, John Wiley & Sons, Inc., 1984.

[BROBST 87]
Brobst, S. A., "Organization of an Instruction Scheduling and Token Storage Unit in a Tagged Token Data Flow Machine," *Proceedings of the 1987 International Conference on Parallel Processing*, v. 3. August 1987.

[DAVIS 79]
Davis, A. L., "A Dataflow Evaluation Sysem based on the Concept of Recursive Locality," *Proceedings of the AFIPS NCC*, v. 48, June 1979.

[DENNIS 80]
Dennis, J. B., "Data Flow Supercomputers," *Computer*, v. 13, November 1980.

[DENNIS 83]
Dennis, J. B., and Rong, G. G., "Maximum Pipelining of Array Operations on Static Data Flow Machines," *Proceedings of the 1983 International Conference on Parallel Processing*, v. 3, August 1983.

[ECOS 89]
AT&T Technologies, Report Alpha 890301-01, *ECOS Workstation User Manual*, AT&T Bell Laboratories, 1 March 1989.

[EVANS 82]
Evans, D. J., *Parallel Processing Systems*, Cambridge University Press, 1982.

[GURD 85]
Gurd, J. R., Kirkhame, C. C., and Watson, I., "The Manchester Prototype Dataflow Computer," *Communications of the ACM*, January 1985.

[HO 83]
Ho, L. Y., and Irani, K. B., "An Algorithm for Processor Allocation in a Dataflow Multiprocessing Environment," *Proceedings of the 1983 International Conferenceo n Parallel Processing*, v. 3, August 1983.

[KARP 66]
Karp, R. M., and Miller, R. E., "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM Journal of Applied Mathematics*, v. 14, No. 6, November 1966.

[KOREN 83]
Koren, I., and Silberman, G. M., "A Direct Mapping of Algorithms Onto VLSI Processing Arrays Based on the Data Flow Approach," *Proceedings of the 1983 International Conference on Parallel Processing*, v. 3, August 1983.

[KUNG 85]
Kung, S. Y., Whitehouse, H. J., and Kailath, T., *VLSI and Modern Signal Processing*, Prentice-Hall, Inc., 1985.

[LEE 87]
Lee, E. A., and Messerschmitt, D. G., "Static Scheduling of Synchronous DataFlow programs for Digital Signal Processing," *IEEE Transactions on Computers*, v. C-36, No. 1, January 1987.

[LEE 89]
Lee, E. A., and others, "Gabriel: A Design Environment for DSP," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, v. 37,No. 11, pp. 1751-1762, November 1989.

[LEE 90]
Lee, E. A., and Bier, J. C., "Architectures for Statically ScheduledDataflow," *Journal of Parallel and Distributed Computing*, v. 10, pp. 333-348,December 1990.

[MOSCOVITZ 90]
Moscovitz, H. S., Yao, K., and Jain, R., *VLSI Signal Processing, IV*, Institute of Electrical and Electronics Engineers, 1991.

[MUNDELL 81]
Mundell, K. J., Linder, M. W., and Conry, S. E., " Processor Allocation in Data-Driven Systems - Two Approaches," *Proceedings of the 1981 International Conference on Parallel Processing*, August 1981.

[PARHI 88]
Parhi, K. P., and Messerschmitt, D. G., "Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding," *IEEE Transactions on Computers*, v. 40 No. 2, 2 February 1991.

[PGMTUT 90]
Naval Research Laroratory, *Processing Graph Method Tutorial*, 8 January 1990.

[POHL 89]
Pohl, I., *C++ For C Programmers*, The Benjamin/Cummings Publishing Company, Inc., 1989.

[POPS 90]
AT&T Technologies, Report 5885401, *Enhanced Modular Signal Processor (EMSP) Principles of Operation (POPS)*, AT&T Bell Laboratories, 20 March 1990.

[PRIMLIB 90]
AT&T Technologies, Report 5885404, *AN/UYS-2 Graph Primitives library - Floating Point*, AT&T Bell Laboratories, 17 September 1990.

[RAU 82]
Rau, B. R., Glaeser, C. D., and Picard, R. L., "Efficient Code Generation for Horizontal Architectures: Compiler Technique and Architectural Support," *Proceedings of the 9th International Symposium on Computer Architecture*, 1982.

[RICE 90]
Rice, M. L., "The Navy's New Standard Digital Signal Processor: The AN/UYS-2," paper presented at the Association of Scientists and Engineers 27th Annual Technical Symposium, 23 May 1990.

[SAWKAR 83]
Sawkar, P. S., Forquer, T. J., and Perry, R. P., "Programmable Modular Signal Processor - A Data Flow Computer System for Real Time Signal Processing," *Proceedings of the 1983 International Conference on Parallel Processing*, v. 3, August 1983.

[SHUKLA 90]
Shukla, S. B., *On Parallel Processing for Real-Time Artificial Vision*, Ph. D. Dissertation, North Carolina State University, June 1990.

[SHUKLA 91]
Shukla, S. B., and Agrawal, D. P., "Scheduling Pipelined Communication in Distributed Memory Multiprocessors for Real-time Applications," *The 18th Annual International Symposium on Computer Architecture*, May 1991.

[STROUSTRUP 86]
Stroustrup, B., *The C++ Programming Language*, Addison-Wesley Publishing Company, 1986.

[STROUSTRUP 90]
Stroustrup, B., and Ellis, M. A., *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company, 1990.

[WATSON 82]
Watson, I., and Gurd, J. R., "A Practical Data Flow Computer," *Computer*, v. 15, February 1982.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Lieutenant Commander Steve Kasputis Department of the Navy Naval Sea Systems Command (PMS 412) Washington, D. C. 20362-5101	1
4.	Mr. David Kaplan Commander of Naval Research Laboratory 4555 Overlook Avenue S. W. Washington, D. C. 20375-5000	1
5.	Mr. Richard Stevans Commander of Naval Research Laboratory 4555 Overlook Avenue S. W. Washington, D. C. 20375-5000	1
6.	Mr. Jerome L. Uhrig, WH 46243 American Telephone and Telegraph Bell Laboratories 67 Whippany Road P. O. Box 903 Whippany, N. J. 07981-0903	1
7.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
8.	Professor Shridhar Shukla, Code EC/Sh Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1

9. Professor Chyan Yang, Code EC/Ya 1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, CA 93943-5000

10. Professor Amr Zaky, Code CS/Za 1
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943-5000

Thesis
L6915 Little
c.1 A technique for predictable real-time execution in the AN/UYS-2 parallel signal processing architecture.

Thesis
L6915 Little
c.1 A technique for predictable real-time execution in the AN/UYS-2 parallel signal processing architecture.

3 2768 00037031 6